EFFICIENT INTEGRATION OF SOFTWARE COMPONENTS FOR SCIENTIFIC SIMULATIONS

Software component interoperability in the creation of flexible scientific simulation environments.

Roman Putanowicz

A thesis submitted for the degree of Doctor of Philosophy (Ph.D.)

Structural Engineering Computational Technology (SECT) Research Group

Department of Mechanical & Chemical Engineering

Heriot-Watt University

Edinburgh, U.K.

August, 2007

Abstract

The advances in multi-physics and multi-scale scientific simulations are the incentive for research in new ways of handling the development of complex simulation codes. The paradigm of component programming and Grid computing set a new level of requirements for simulation codes in terms of the interaction between highly heterogeneous components and the adoption of new codes. This thesis stresses the need for the development of software integration techniques for scientific simulation codes. Ensuring component interoperability allows not only the building of more powerful programs but, as it is strongly stressed in this thesis, helps to lower the cost of the verification and validation of simulation programs.

This thesis introduces the notion of a hybrid simulation system as a system consisting of generic system programming language libraries, scripting language interpreter, interface modules between scripting language and system language components and interface generation tools. It is argued that hybrid are to be the most appropriate environment for the development of academic simulation codes. The main contribution of this thesis is the idea of Grid and Geometry Exchange Services (GAGES) as an example of a hybrid system in the domain of pre- and post-processing of scientific simulations. The case studies undertaken to support claims about GAGES and hybrid systems yielded several practical results, for instance an anisotropic mesh generator, a surface mesh generator, a grid plotting library and new tools for multi-language programming. A posteriori analysis of the development efforts resulted in another original idea of the usage of a SWIG compiler interface specification as a universal scientific interface description language.

To my wife and son.

Acknowledgements

I would like to gratefully acknowledge the support of the Overseas Research Students Award Scheme and James Watt Scholarship programme which made it possible for me to study at Heriot-Watt University.

I would like to cordially thank my supervisor Prof. B.H.V. Topping for his invaluable guidance and effort to keep me right on the track. He has been also the most patient person in explaining the rules of the scientific development to me.

During my stay at Heriot-Watt University I enjoyed many valuable discussion and received support from Dr. Peter Iványi, Renata Obiała, Jelle Muylle and Janos Nezo.

I am also in dept to Dr. Frédéric Magoulès for his constant encouragement and friendship.

I would like to thank Dr. Guntram Berti for his help in studying the `GrAL` library and for making it freely available.

I would like to thank to all the people from the Institute of Computational Civil Engineering Cracow that have helped me in my work. I express a special thanks to Dr. Aleksander Matuszak who has been my computer guru since I started my work and to Prof. Zenon Waszczyszyn and Dr. Jerzy Pamin for creating the environment for me to finish my work.

I would like to express my gratitude to my parents for their support and concern for me and to Krzysiek for being such great brother.

To my wife Magda and son Michał – without your love and support this work could not be done.

ACADEMIC REGISTRY
Research Thesis Submission

Name:	Roman Putanowicz		
School/PGI:	Engineering and Physical Sciences		
Version: *(i.e. First, Resubmission, Final)*	Final	Degree Sought:	PhD

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1) the thesis embodies the results of my own work and has been composed by myself
2) where appropriate, I have made acknowledgement of the work of others and have made reference to work carried out in collaboration with other persons
3) the thesis is the correct version of the thesis for submission*.
4) my thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying, subject to such conditions as the Librarian may require
5) I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.

* *Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is submitted.*

Signature of Candidate:	Roman Putanowicz	Date:	09.08.2007

Submission

Submitted By *(name in capitals)*:	~~ROMAN PUTANOWICZ~~ JONATHAN SHEPPARD
Signature of Individual Submitting:	~~Roman Putanowicz~~ J.D.Sheppard.
Date Submitted:	~~09.08.2007~~ 5/10/2007

For Completion in Academic Registry

Received in the Academic Registry by *(name in capitals)*:	VAL MURDOCH		
Method of Submission *(Handed in to Academic Registry; posted through internal/external mail)*:	By hand		
Signature:	Valerie A Murdoch	Date:	5/10/07

Contents

List of Figures

List of Tables

List of Publications

[1] F. Magoules and R. Putanowicz. Visualisation of large data sets by mixing Tcl and C++ interfaces to the VTK library. *Computers & Structures*, in press (2006).

[2] F. Magoules and R. Putanowicz. Large-scale data visualization using multi-language programming applied to environmental problems. *International Journal of Energy, Enviroment and Economics*, 13(5):45–75, 2006.

[3] F. Magoules and R. Putanowicz. Optimal convergence of non-overlapping Schwarz methods for the Helmholtz equation. *Journal of Computational Acoustics*, 13(3):525–545, 2005.

[4] R. Putanowicz. Grids manipulation environment in Python In A. Garstecki, B. Mochnacki, and N. Sczygiol, editors, *Proc. CEACM Conf. on Computational Mechanics and 16th Int. Conf. on Computer Methods in Mechanics CMM-2005* Czestochowa, Poland, 2005. Paper on CDROM (8 pages).

[5] F. Magoules and R. Putanowicz. Non-overlapping Schwarz methods for the Helmholtz equation and related shape optimization problems. In S. Domek and R. Kaszynski, editors, *Proceedings of the 10th IEEE International Conference on Methods and Models in Automation and Robotics*, Miedzyzdroje, Poland, 30 August – 2 September 2004, volume 1, 2004.

[6] B.H.V. Topping, J. Muylle, P. Iványi, R. Putanowicz, and B. Cheng. *Finite Element Mesh Generation.* Saxe-Coburg Publications, Edinburgh, 2004.

Chapter 1

Introduction

The basis of this thesis is the author's experience that programming large applications is difficult. The author has usually written small programs with well defined specifications, where the domain analysis was simple and design not complicated. Such programs were usually good exercises for mastering the programming language skills but left the author ill-prepared when faced with the task of writing larger systems. The success of writing a large system crucially depends on a good understanding of the area of application, careful design, and the subdivision of the problem into smaller parts with well defined interfaces between them. Frequently, a researcher will generally investigate the problem, design the software and implement it, all at the same time. Frequently the time spent on system design is ridiculously little, while the success of a software project is a direct derivative of good design.

The author witnessed the building of a machine. After trying to assemble its parts, it turned out that a special custom type of a wrench had to be made, because the designer forgot to leave a maintenance space. Another, similar example, was when the assembled and working machine was about to leave the production hall, and was found that a machine part sticks outside the door outline. The machine could not be disassembled so a piece of the wall had to be cut out. Engineering practice is full of similar examples and software engineering is by no means an exception. It is probably subjected to more flaws of that kind, taking into account

the abstract matter it deals with and its complexity. The author was motivated by a "design bug" (not taking into account programming bugs) to start thinking about some more general and radical solutions other than a trial and error process.

In software engineering the common way to utilise good design is to encapsulate it into libraries. This is similar to using off-the-shelf components such as nuts, bolts, springs, etc. in mechanical engineering. However, the mechanical components are highly standardised and are meant to fit together, while scientific libraries can not be so easily used together. Most of the scientific libraries use a similar set of abstractions – vectors, matrices, lists, sets, curves, meshes, etc., yet there are usually incompatibilities between libraries. It is like having a lot of suppliers providing nuts and bolts, but each supplier's specification differs.

The work on this thesis was fuelled by the dream of having something that can be seen as 'Lego bricks' for computational scientists. The real Lego bricks come in an abundance of colours and shapes to permit the building of almost any kind of structure. Yet, they are based on a principle that it must be possible to connect any two bricks in one or another way. The equivalent 'Lego' for computational scientists would be a set of software components that can be linked together and/or used interchangeably. The components encapsulate various abstractions of the application area, thus permitting programming not in terms of variables and functions, but directly in terms of concepts used to describe the problem.

The search for such bricks brought the author's attention to the concept of Problem Solving Environments (PSE) [67] and component programming [94].

A problem solving environment is a collection of tools together with a linking mechanism, for example a scripting language interpreter, that are sufficient to solve a given class of problems, for instance the analysis of membrane structures or the convection dominated flow of pollutants. PSE also contains ways to incorporate novel solution methods and permit problems to be expressed in a language close to the application area.

Component programming in turn is based on reusing component abstractions. It might be thought of as a generalisation of object oriented programming, in the sense, that it also deals with encapsulation of states and behaviours inside some entities, here components. However, components are usually created at much coarser level than objects, are distributed in binary form, take into account language and operating system dependencies and allow for more detailed introspection.

This thesis builds prototypes of several components of a PSE for a small subset of computational mechanics, mostly related to preprocessing and postprocessing tasks. The aim of this thesis is to identify the main difficulties and challenges in building full scale PSEs and to develop new technologies necessary to overcome those difficulties. The second aim is to provide enough empirical knowledge for further study and a methodological investigation of component programming for PSEs in computational mechanics.

1.1 Some remarks concerning users of simulation environments in academia

As it is not possible to design a car which will satisfy all drivers, so it is not possible to build a simulation environment that will suit all users. This is rather a trivial observation, but it is important to state it here, before jumping into the presentation of specific techniques and solutions.

Firstly, it should be kept in mind that all the solutions presented have their areas of applicability, and that they are not 'silver bullets'. The 'silver bullet' remark could be probably made for any field of study but the marketing hype accompanying new computer technologies often makes users forget this. Examples of this are the introduction of object oriented programming[1], object data bases, Java, or the recent

[1]One should rather say C++ programming as object oriented techniques precede the introduction of C++.

enthusiasm concerning XML technologies or Grid computing. In their time, these technologies were advertised as a nearly universal panacea, to later turn out, that each of them has their own downside. Somehow related to this is the famous saying of Bill Gates that nobody needs more than 640 kB of RAM. It is because computer science and the computer industry are one of the most dynamically developing fields in the last 50 years.

Secondly, computer users can get very emotional over the software tools they use, starting from the choice of operating system, through programming language, to the way of indenting source code. Examples of this can be various flame-wars held on the Internet between user communities.

The above is said to stress that the discussion of the merits of computer(software) technologies should not be stated in absolute terms but in the context of a particular application or a particular problem to be solved.

The context of the computer technology application can be sketched from the character of user groups. This thesis is concerned with building scientific simulation environments. Moreover it is mostly concerned with the simulation of physical systems as opposed to simulations in mathematics (e.g. in computational algebra) or in biology (e.g. human genome research). Further, it is mostly confined to users placed in academia. But even then, such a restricted user group is too diverse to be characterised in a useful way. For the purpose of further discussion the above group of users of simulation codes in academia will be divided into three categories. The criteria to establish these categories are:

- the aim of performing simulations,
- the usage of commercial products, and
- the available resources in terms of manpower and money.

Two cautionary remarks must be made before proceeding any further. Firstly, the presented distinction is not based on any objective research but just on the au-

thor's observations which might be biased. Secondly, the distinction is not a crisp categorisation but it is a fuzzy classification and the categories surely overlap.

The first category are users associated with world class computer laboratories such as CERN, Sandia Laboratories, INRIA, etc. The aim of their research and computer simulations is the breakthrough in science, e.g. thermal fusion. Such laboratories need to develop their own specialised, efficient codes, which are run on dedicated computer installations. Such codes are the cutting edge of simulation software. Such laboratories are most often supported by governmental or international agencies.

The second category of users are the ones associated with computer laboratories routinely cooperating with industry. The aim of their simulations is the solution of practical, though often nonstandard, engineering problems. The tools used for the simulations must be robust and fast, as often the results are used to take decisions involving human safety. The code development is restricted to enhancing or connecting fully fledged commercial products. Such laboratories are mostly supported from research and development contracts from industry.

The third category are users associated with computer laboratories having an almost purely academic character. The aim of their simulations is gathering insight and dissemination of knowledge. They educate specialists forming the two other groups. The use of commercial codes is restricted because they are not necessarily the best educational tools and because they are not cost effective. Such laboratories often spend considerable effort on developing custom simulation codes, as this gives the most insight, and allows unrestricted dissemination of software and results. Such laboratories are supported from research grants and resources allocated for education.

From the above it can be seen, that each group has rather diverse requirements concerning software development. Though, in principle, they are aiming for the same – the science, however they definitely go along different paths. Such differ-

ences should not be neglected, especially if one cares about effective management of scientific work.

1.2 An outline of the thesis

This dissertation is written from the stand point of a member of the third category of the user groups.

The motivation is highly utilitarian, namely an attempt to answer the question: what can be done, taking the current state of technology and available tools, to make the development of simulation codes easier?

The first general claim is that attention should be directed towards component integration. Advances in telecommunication technologies have opened the whole world of possibilities, but are also forcing us to live with inherent heterogeneity. Research projects that span multiple fields or multiple physical scales will need to deal with different operating systems, languages, algorithmic requirements or just human preferences.

The second general claim is that it is not possible to reach uniformity in terms of programming languages. The appearance or the rising popularity of various scripting languages and their diffusion to the area of scientific simulations, traditionally reserved for `FORTRAN` or C, is one part of the evidence on which this claim is based.

The third issue and the central point of this thesis is related to handling geometry and grid[2] data structures. They are the building blocks for the preprocessing and postprocessing phases for the simulation. These phases are constantly becoming more and more important as we try to closely match reality. In order to integrate various geometric and grid based data structures, this thesis proposes to use original concepts of geometry and grid buses. The buses are the mechanisms to interface various programs and libraries. Contrary to other proposals, which are centred around data format descriptions, this thesis postulates the building of geometry and grid

[2]In this dissertation the terms "grid" and "mesh" will be used interchangeably.

buses on the basis of APIs of some highly generic libraries. It is also claimed that this is the fastest way to obtain working component integration solution. This postulate is further refined into a four-tier layered structure of geometry and grid buses. Also, the thesis postulates that scripting languages are an indispensable part of any practical integration solution, as they allow a balance between the development and the code speed and flexibility.

One important observation pertains to mesh generation. It is shown that this is a major component as it links geometries and grids, and that there is still to few freely available tools, especially in the area of surface mesh generation. A simple remedy for this is also proposed in Chapter 9.

The first two general claims are supported on the basis of the background literature and references presented in Chapters 2, 3 and 4. Refinement of the thesis related to geometry and grid buses is presented in Chapter 5. The remaining chapters except Chapter 11, present practical and detailed case studies, which provide specific examples to demonstrate the main thesis of the dissertation.

It is assumed that claims about geometry and gird busses or about the role of scripting language interfaces cannot be truly supported in any but an empirical way. It makes the research a painstaking effort as one has, for scientific objectivity, to try, at least on a basic level, competing software solutions.

Such an empirical study puts a researcher in a realm of software bugs, missing documentation, unfinished projects and the frustration of the discovery of another dangling pointer in a code. This is however the real world, and software engineering, contrary to more mature engineering fields, which are turned into soulless routine, still has the charm of pioneering times. One of the visible aspects of this is that, despite enormous effort, software engineering very slowly permits codification and standardisation.

1.2.1 Thesis objectives

The thesis objectives are:

1. To design an effective methodology and architecture for connecting components of simulation systems in computational mechanics, especially the ones based on the finite element method. The proposed methodology should take into account the diversity of user groups and balance the efficiency of programmers developing software versus the efficiency of the software itself.

2. To introduce the notion of a hybrid system as a software package consisting of four basic components:

   A – specialised libraries written in system programming languages, compiled for efficiency,

   B – scripting programming language (or languages) providing an interpreted environment,

   C – interface modules providing a bridge between specialised libraries and the scripting language environment,

   D – tools for creating new interface modules for the user's custom codes.

   The aim of this thesis is to show that such hybrid systems play an important role in component integration, as they overcome difficulties related to multi-language programming, and effectively balance programs' development speed versus execution speed, by enabling integration to be done at the scripting language layer or system language layer.

3. To build a concrete framework based on the proposed architecture. The goal of this framework is to link specialised tools for finite element pre- and postprocessing. The framework will be built using a generic programming paradigm for handling geometric descriptions and mesh data structures.

4. To show feasibility of the proposed methodology by connecting concrete instances of pre- and postprocessing tools with the framework mentioned in objective 3.

1.2.2 Methodology

Software integration can be achieved on various different levels starting from the level of single procedures, through object and classes, and ultimately incorporating whole complex components.

In this thesis the integration based on the most coarse, component level is taken as the starting point. This is motivated by the observation that integration on the component level is the most common situation when building simulation systems from the available building blocks. Integration on the finer level, for instance on the level of objects and classes would be more feasible for a system built from scratch, where one has more freedom to shape system elements in a desired way.

The proposed architecture will be analysed from the perspective of two orthogonal points of view as shown in figure 1.1.

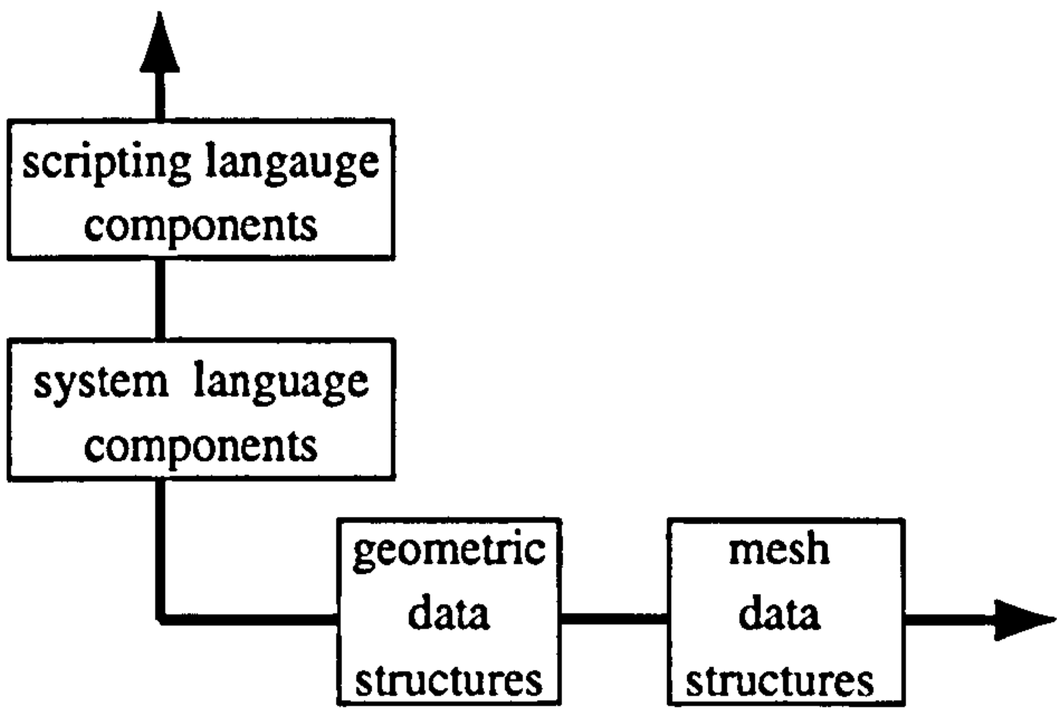


Figure 1.1: Two orthogonal views on the proposed architecture.

One point of view concentrates on the transformation of data from geometric to mesh description. This transformation appears naturally when solving problems with finite element method, and taking this perspective into account will allow us to

ensure that the proposed architecture can support a full chain of processing tasks.

The other point of view is the transition from system programming languages to scripting programming languages. In other words, a concrete data processing tasks, for instance mesh generation, can be expressed and controlled by means of scripting languages or system languages. This perspective allows us to balance ease of systems development with execution efficiency.

In order to support statements about the feasibility of the proposed methodology several detailed case studies will be presented. These case studies will show how to integrate concrete components, for instance visualisation or mesh partitioning, with the proposed framework, and will also show how to build a scripting language interface to them. The case studies are selected in such a way as to possibly cover the full range of processing tasks: geometry design, mesh generation, mesh manipulation and visualisation. They are also selected to show distinct characters of components: programs versus libraries, different data models (structured and unstructured meshes, for instance), different programming languages, etc. Although each case study could be treated as a separate research project, together they give a general picture of the proposed system

Chapter 2

The need for software integration

This Chapter shows the need for data and algorithm exchange mechanisms for computational simulations. Though it may seem redundant to stress that need over and over again, as it may seem widely recognised, there are at least two reasons to raise that problem once again. The first one is that we still lack widely accepted solutions, especially in academic research [18]. The second is that in the light of recent advances in information and telecommunication technology during the last few years and the resulting progress in computational simulations, exchange of data and the integration of simulation software becomes a primary issue [59, 58].

The subsequent sections firstly give some brief overview of advances and trends in computational science. Then some problems related to data exchange like verification and validation of simulations, integration of simulations of different phenomena are identified and described. Next, some reasons for the unsatisfactory state of affairs in the verification and validation of most efforts in computational science is linked with insufficient development of mechanisms for the exchange of simulation data. Finally the benefits of the development of standard mechanisms for exchange of simulation data and a vision of the application of such mechanisms is given.

2.1 Trends in computational simulations

We are witnesses of the enormously fast development of hardware, software and communication technologies. However in many cases, we are so accustomed to the pace of the changes, that we take them for granted. Many computer users see it as obvious that their computer equipment becomes "obsoleted" in approximately every two years and they "have to" change it for a new one. In the case of software optimisation we can read [90] that it might be cheaper to do nothing and wait a few months for advances in hardware, which will render our optimisation problem nonexistent (this statement can be much argued about, but important is the fact that such statements appear at all). The above examples are the byproducts of a more general phenomenon called Moore's law [85, 86]. Moore's law states that every 18 months the computing power at our disposal doubles (which relates to increasing density of transistors). So far the prediction was quite accurate though we can now hear claims [80] that we are quickly reaching the end of the applicability of Moore's law.

The other area of rapid changes is communication technology. Again, though we take as obvious the enormous spread of the Internet, the development of cellular phones, wireless networks and the merging of various communication channels (cellular phones and computer networks for instance) the changes have a profound effect on almost all aspects of our life. In [87] Turk calls those changes "the digital revolution" and puts it in one row with other communication revolutions: the writing revolution (invention of writing), the paper and print revolution, and the electronic revolution (invention of telegraph, telephone, radio and television).

The advances in information and communication technologies have their impact on science and engineering. An in depth analysis of that impact on computational science is beyond the scope of this Chapter, but a discussion of that topic can be found for instance in references [66, 48, 88] and especially in references [59, 58]. For the purpose of future discussion two issues will be examined more closely: simulation

of multi-physics phenomena and distributed collaborative research.

2.1.1 Multi-physics simulations

In situations, when the physical or mathematical models and/or computing power at our disposal is not enough to handle a simulation of a complete complex phenomenon, the natural approach is to consider in a detailed way only one dominant aspect of that phenomenon (e.g. the mechanical, thermal or chemical processes) and to neglect the others, or to handle them in an approximate or averaged way. However, with advances in physical and mathematical modelling and primarily with increase of computing power, it becomes feasible to perform an in depth analysis of multiple physical phenomena at the same time, capturing their essential interactions. What is more, it becomes not only feasible but highly necessary to perform such simulations to either provide the required level of safety in hazardous environments (e.g. in space exploration) or to cut the cost and impact on the environment (e.g. the US ASCI project – the Accelerated Strategic Computing Initiative, uses numerical simulations instead of trial nuclear explosions) or to provide more competitive products. It is not even necessary to reach for "high science, high technology" examples. A good example from everyday engineering practice, where such an analysis is indispensable, is the case of metal casting [89]. Analysis of metal casting processes involves mechanical, thermal and fluid flow modelling as the liquid metal fills a mould and cools and solidifies. Additionally, liquid metal can be stirred and controlled by an electromagnetic field, and that accounts for several other phenomena to be modelled.

As the engineering tasks become more complex and greater precision, speed, and efficiency of engineering structures and processes is required, then multi-physics simulations turn from a possibility to a necessity [58].

2.1.2 Distributed collaborative research

The second issue, central to further discussion, is the emergence of distributed collaborative research. Innovations in communication technologies have several repercussions. First of all, they enable sharing resources – both in terms of data and processing power, as well as human resources. Moreover, those distributed resources (distributed in geographical and institutional sense) can be conveniently reached through interfaces (such as the World Wide Web) which hide their distributed nature and complexity. Magnified computational power not only allows the solution of larger problems in one field, but also combines several distinct efforts. Integration of efforts becomes a necessity as we tackle more and more complex problems. In the case of simulation software Epperly et al. [28] say: "As simulations become increasingly sophisticated and complex, no single person – or even single institution – can develop scientific software in isolation. Development teams rarely poses sufficient resources and scientific expertise in all required domains to successfully create a complex application from scratch. Instead, physicists, chemists, mathematicians and computer scientists concentrate on developing software in their domain of expertise. Computational scientists create simulations by combining these individual software pieces."

Collaboration enabled by modern communication channels is not only opportunity but in many cases a necessity. Engineering design is an example of this. Reference [114] states that: "Fundamentally, design activities are not isolated activities. General observations developed over the years by engineering design researchers confirm this statement. Leifer [152] claims as a first rule: "(...) design is a social activity. A social activity implies many types of interactions and communications among the actors of the design. Improving the quantity and quality of interactions among actors in design teams improves the quality of the design."

Crystallisation of the described themes can be seen in two "hot" research topics: problem solving environments (PSE) and Grid Computing.

2.1.3 Problem solving environments

A short definition of a PSE is taken from [67]: "A PSE is a computer system that provides all the computational facilities necessary to solve a target class of problems. These features include advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems, so users can run them without specialised knowledge." Problem solving environments can be build using various architectures, but in the light of providing the simulation system with a sort of "ingelligence", the blackboard architecture looks quite interesting. Reference [159] provides general overview on the blackboard architecture and reference [158] shows an example of the usage of blackboard architecture to the building of a simulation system based on the finite element method and using object-oriented paradigm.

Though the concepts of the PSE have been available for more than thirty years [68] only the recent advances in computer hardware and software have made the PSE feasible. Problem solving environments do not automatically enable collaborative research, though there is also strong interest in Collaborative Problem Solving Environments. However, it is rather obvious that designing and building a PSE require collaboration of scientists and engineers in various fields.

2.1.4 Grid computing

One of the very rapidly developing research areas is "Grid Computing". There is in fact so much hype about the term Grid [132] that it has been used (and sometimes abused) in a plethora of expressions: "Data Grids", "Knowledge Grids","Tera Grids", "Scientific Grids", "Grid Computing", etc.

For the purpose of this dissertation the definition presented in [10] will be used: *"Grid is a type of parallel and distributed system that enables sharing, selection, and aggregation of geographically distributed 'autonomous' resources dynamically at*

run time depending on their availability, compatibility, performance, cost, and users' quality-of-service requirements."

In [132] Foster provides a simple checklist capturing the basic Grid characteristics. According to that checklist a Grid is a system that:

- coordinates resources that are not subjected to a centralised control,
- uses standard, open, general-purpose protocols and interfaces
- delivers nontrivial quality of services.

One may wonder how Grid architecture is different from cluster computing, single parallel systems, P2P (point-to-point) computing or web services? This question is discussed for instance in [30] where authors argue that: "the Grid concept is indeed motivated by a real and specific problem and that there is an emerging, well defined Grid technology base that addresses significant aspects of this problem." Grid computing is distinguished from the conventional distributed computing such as dedicated parallel systems and cluster computing by the lack of a centralised management or the single ownership of computational resources. Grid allows and promotes creation of dynamic "virtual organisations", whose participants share their resources. Grid systems also differ from web services in the sense that web services can be used as a fabric from which Grid systems are built.

Computational Grids enable a new scale of computing. Citing reference [10]: "When computational processes are endowed with the ability to be aware of both their needs and the computational world around them, and when this computational world – known as the Grid – is endowed with the ability to support such dynamic applications, we will have reached an entirely new stage in computing." The researchers envisage that Grid systems will allow:

- fault tolerant computing – if one computing node fails or cannot deliver satisfactory performance, the computational process can migrate to other available nodes,

- awareness of needs and resources – the computing process can spawn number of routines and run them asynchronously on available machines,

- instant visualisation of results as computing progresses,

- monitoring and steering computations from any device (desktop computer, wireless hand held devices, mobile phones),

- rapid building of virtual organisations with computing resources necessary to quickly simulate complex events (e.g. in case of emergency like nuclear reactor failure, chemical contamination spread, etc.).

References [10, 107] contain further details of a vision of the new brave world that Grid technology will enable.

2.2 Characteristics of complex simulation systems

The two research areas outlined above share some common characteristics: communication between distinct components, heterogeneity of components and the requirement to dynamically adopt new components.

2.2.1 Communication between distinct components

The need of communication between distinct components is especially evident in the case of Grid computing, where distinct nodes providing data, data storage, processing power and visualisation capabilities are connected through the network. However, also problem solving environments are built from distinct components. It is almost impossible to build any but a toy PSE from scratch, as a monolithic architecture. In order to make a PSE feasible, one has to use existing software – independently built packages for specialised tasks – linear and nonlinear algebraic solvers, PDE solvers, mesh generators, domain decomposition packages, and so on.

Ensuring that these distinct components can be linked is not an easy task, and this thesis proposes a methodological approach for building such links.

2.2.2 Heterogeneity of components

Heterogeneity and requirements for interoperability are most visible in the case of Grids, where by default one deals with nodes based on different hardware, operating systems, etc. In the case of PSEs even if they are running on single hardware under a single operating system their components can be based on different design principles and implemented in multiple programming languages. Additionally, the scope of grid computing and the PSE introduces heterogeneity at a much higher abstraction levels – one can imagine simulation involving discretization using finite elements and finite volumes or geometry description via B-Rep (boundary representation) and CSG (Computer Solid Geometry) not to mention exact mathematical descriptions and their numerical approximations. By the very nature PSE and grid computing encourages the use of a wide variety of high level abstractions instead of enforcing a single approach. Because these high level abstractions are most conveniently delivered as components, this suggests us that, from the point of view of development efficiency, we should look at the problem of linking components as the whole entities.

2.2.3 Dynamic adoption of components

Dynamic adoption of new components is the very basic characteristic of Grid computing systems. Grid computing systems are by definition very flexible and fluent. For instance, if during the simulations it turns out that some software component (e.g. finite element package) is unable to carry out the simulation or is inefficient, the simulation can be reconfigured to use another software package in the subsequent analysis. That software must be dynamically incorporated into the computing system and used sometimes without even stopping the simulation. Problem solving environments are not so dynamic in nature, but they also benefit from the freedom

of merging various components. In fact, component programming [94] – a programming paradigm based on assumption of building programs from configurable, standardised binary components, is one of the fundamental technologies that PSEs are based on. For instance, one of the assumed features of PSEs is that they will allow automatic selection of the appropriate solution method – such selection is much easier when solver modules are encapsulated in easily pluggable components.

2.3 Verification and validation as a drive for software integration

2.3.1 Software verification and validation

Another very strong drive for software integration is the necessity for verification and validation (V&V for short) of simulation codes. The classical scientific method depicted in Figure 2.1 is based on a process of observation, hypothesis development, experimental design, hypothesis test and iterative improvement [1]. Introduction of computers and numerical simulations brought new element to this scheme, as depicted in Figure 2.2. This is well known Sergent Circle [1]. Though it might be questioned [108] if it precisely captures the scientific process, it clearly indicates that the two activities become an integral and essential part of it:

- validation process – determination to which degree a computer simulation is an accurate representation of the real world,
- verification process – determination to what extent computer simulation correctly represents the conceptual model and its solution.

Without the two processes becoming an integral part of any scientific development the results obtained are at least questionable.

Despite making a gross simplification, one can say that verification and validation is based on a comparison of data produced by the examined software with data from

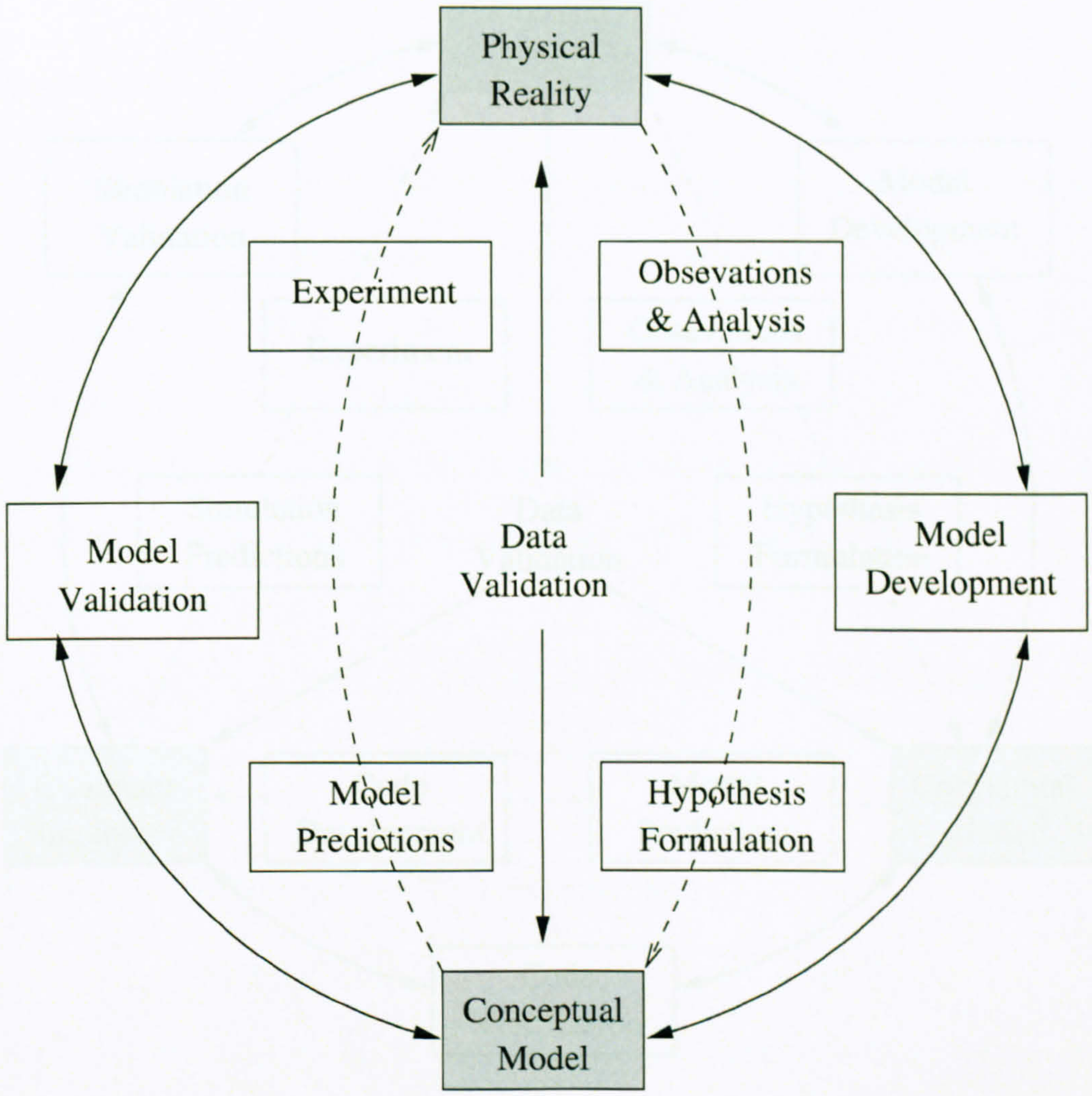


Figure 2.1: Illustration of traditional scientific method [1].

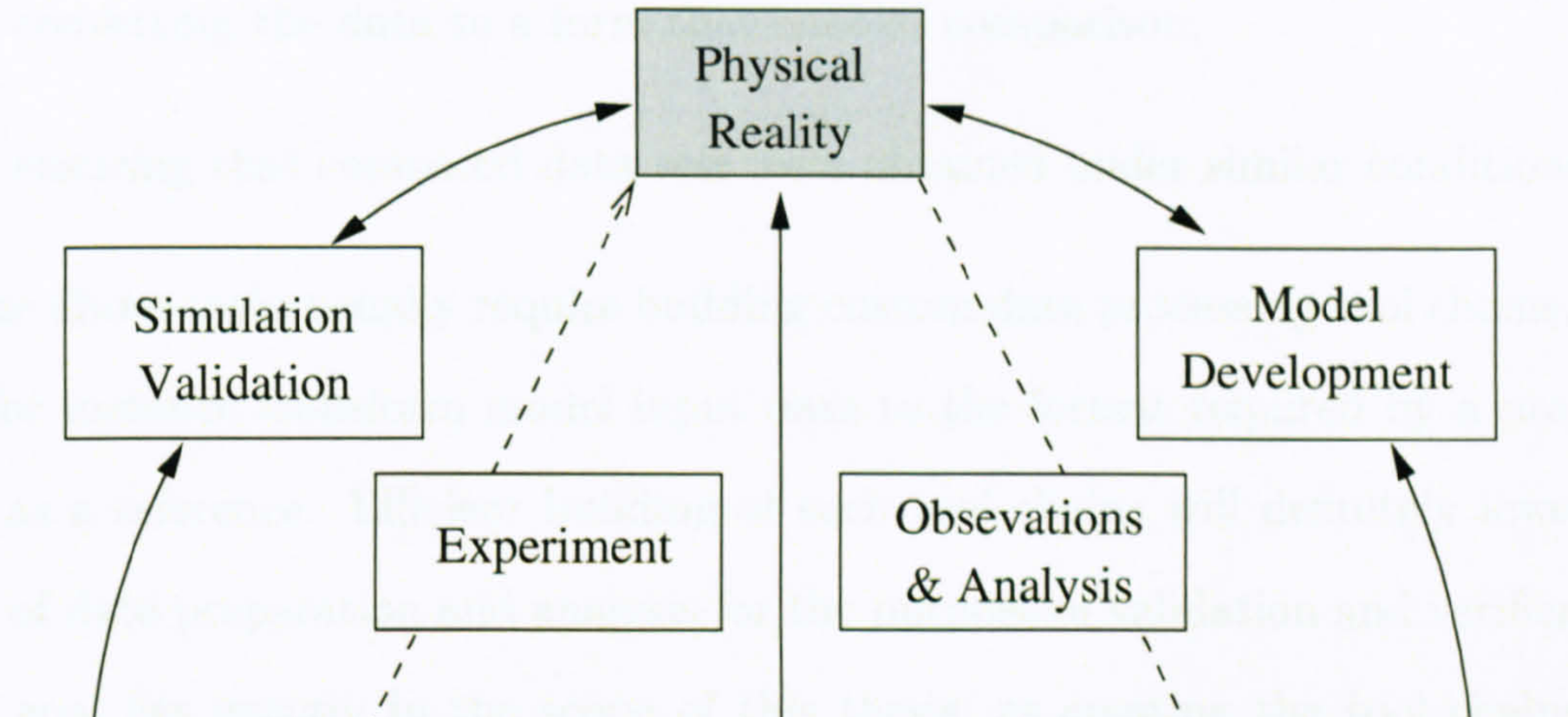


Figure 2.2: Illustration of the scientific method taking into account computer simulations [1]. This is so called the Sergent Circle.

real world measurements, data produced by examination of mathematical models or data obtained by using similar software. This can be reduced to three basic problems:

- obtaining comparison data,
- converting the data to a form that enables comparison,
- ensuring that compared data sets were obtained under similar conditions.

All the above tasks usually require building custom data processing tool chains, that will for instance transform model input data to the format required by a program used as a reference. Efficient building of such tool chains will definitely lower the costs of data preparation and analysis for the purpose of validation and verification. Such goal lies exactly in the scope of this thesis, as creating the tool chains will require effective integration of various software components. This point is further discussed in section 2.3.3.

Before we go further with the details of validation and verification it is appropriate to consider a related subject close to the heart of every programmer.

2.3.2 How faulty is scientific software?

Every programmer knows that a bug free program is an abstraction that can be achieved in 100% only by some simple programs. Every more complex program contains some bugs, and they can be slowly eliminated one by one, only by running the program several times and observing its behaviour, though without a guarantee that all bugs will show up. Knowing that there are programming bugs one can ask a question how buggy is scientific software? This question was the reason for a study undertaken by Hatton and his coworkers and published in two papers [6] and [40].

In the first four year study, nine commercial seismic data processing packages written independently in the same programming language were analysed. They were calibrated with the same input data and input parameters. It turned out that they

	C	Fortran
Total lines analysed	3,305,628	1,928,011
Number of participating organisations	47	26
Largest package in lines	770,444	431,655
Smallest package in lines	806	361
Average package size in lines	60,102	28,353
Total packages	55	68
Number of different disciplines	20	41
Total executable lines	1,737,536	1,389,712

Table 2.1: Amount of code analysed in Hatton's T-experiment [6].

differed in the first or second decimal place and that the discrepancy is entirely a result of software errors. In extreme cases the results obtained contradicted each other.

In the second study called T-experiment, which consisted of the static (without running the code) and dynamic tests, a large number of C and `FORTRAN` codes were analysed, as shown in Table 2.3.2

This study showed that software accuracy is greatly undermined and that the codes are full of statically detectable errors. This study is best concluded by the following quote from [6]: "Taken with other evidence, the T experiments suggest that the results of scientific calculations carried out by many software packages should be treated with the same measure of disbelief researchers have traditionally attached to the results of unconfirmed physical experiments."

The above indicates that a more systematic approach to program testing should be developed and applied. Enabling more freedom at component composition supports program testing as it allows the components to be run in different environments, thus making it more likely that the bugs will show up earlier in the testing.

2.3.3 Cost of V&V

The very relaxed approach to software verification and validation may be caused by two factors. One is the lack of a proper attitude to computer results, the type of approach expressed by a quote "if calculated by a computer then they must be right". The second are the costs of doing proper V&V. To make sense, both validation and verification must be versatile but this takes time. The first difficulty lies in gathering comparison data. Even if we are fortunate and can obtain the data, it often requires some preprocessing before can be used in a comparison study. This may require writing format converters, interpolation routines and the like. Also, when undertaking software verification by comparison with reference software, quite substantial effort might be required to translate input data from the user system to the format required by the reference system. One can expect an even higher cost when trying to verify or validate published algorithms with one's own algorithms or with one's own data. First of all the publication must contain enough technical details to allow seamless reimplementation and this is seldom the case. Even if we have a good paper and we collect all the pieces, like for example mesh generators, linear algebra library, solvers, visualisation, etc., then there is no guarantee that the pieces will match. Thus in the circumstances of increasing pressure for new, original results, it is likely that the V&V will not gain deserved attention.

How does this relate to software integration? By providing inexpensive and reliable ways of exchanging data sets, one automatically promotes sharing of validation data. By making various tools compatible, one lowers the cost of building V&V tools. Finally by providing matching software components, it is easier to test a particular component in different configurations, thus decreasing the chance that some bugs will slip through the testing net.

2.4 Concluding remarks

In this Chapter the most important trends in computational engineering simulations were analysed. It was shown that in order to advance in the areas of multi-scale, multi-physics simulations and grid computing the new paradigm of the component programming should be fully employed. Three main issues: communication between distinct components, heterogeneity of components and dynamic adoption of new components were highlighted as the ones, which should be carefully considered when selecting a mechanism for the component integration.

It was also pointed in this Chapter that the rising complexity of simulation codes makes the issue of program validation and verification much more difficult. One has to cope not only with internal complexity of components but also has to carefully watch the issues of component interaction and dependencies. Each piece of code is written with a set of assumptions, for instance regarding the transfer of responsibility to free unused resources. These assumptions are not always clearly stated in the code interface or documentation, and accidentally breaking them may lead to unexpected program crashes or worse, make the program to produce nonsensical results. Thus in this Chapter it was stressed how important it is to commonly adopt the software verification and validation practices. It was also argued that by enabling various components to work together one can effectively lower the costs of the systems V&V by simplifying or automating the tedious tasks of data, protocols or APIs translation when preparing the test cases.

In the next Chapter an overview of the techniques for the software integration will be presented. It will be shown that the techniques can be divided into several categories and that in each category there are many tools available to programmers. It is important to realise that each technique or tool has its own restrictions and application costs. The selection of an appropriate technique for component integration is not a trivial task, especially in the light of the requirements stated in section 2.2. In the next Chapter it will be also pointed that the software integration techniques

based on scripting languages are worth considering, especially for the purpose of preparing the tools for supporting the validation and verification tasks.

Chapter 3

Techniques for software integration

When considering a component based system it naturally appears the question about the components integration mechanism. It will be shown in this Chapter that there are many different approaches to the software integration, and many tools supporting a particular approach. The selection of a technique for the software integration is not a trivial task and it is quite difficult to change the component integration mechanism once the simulation system is in an advanced building stage. It should be also noted, that the techniques and tools have their intrinsic usage restrictions as well as the usage costs. Thus the selection of them should be carefully considered.

In this Chapter various software integration techniques will be presented in order to sketch the background on which the particular technique based on the scripting languages will be advocated.

3.1 Achieving interoperability

In the last Chapter it was shown that interoperability is a primary issue for grid computing systems and an important factor for PSEs. This section discusses the different ways in which interoperability can be achieved. It starts by providing the

definitions of protocol, data format, service and API.

Protocol: As suggested by Foster (et al.) [30] a protocol is "a set of rules that end points of a telecommunication system use when exchanging information." An important property of protocols is that they admit to multiple implementations: two end points need only implement the same protocol to be able to communicate. Examples of protocols are: Internet Protocol (IP), Transmission Control Protocol (TCP), Transport Layer Security (TLS) Protocol, Hyper Text Transfer Protocol (HTTP), and Remote Procedure Call (RPC) protocol.

Data exchange format: While the Open Systems Interconnection (OSI) model distinguishes application level protocols as a set of rules for encoding and interpreting information within an application domain and while data exchange formats (such as ASCII, JPEG, HTML, IGES, VRML, HDF, etc) are nothing more than application level protocols, it might be advantageous to treat data exchange formats separately from protocols. Data exchange formats deal with high level abstractions such as meshes, fields, images, tables while protocols are commonly associated with bit streams, packets, frames – generally with the lower level transport layers. Data exchange formats will be thus understood as rules for exchanging or storing complex, domain specific objects. Information encoded in standard exchange format can be transferred using the network protocols. The basic unit for data exchange formats is a file. The applications exchanging data via data exchange formats does not have to share any implementation details provided they correctly interpret the data format.

Service: In reference [30] a service is defined as "a network-enabled" entity that provides a specific capability, for example, the ability to move files, create processes, or verify access rights. A service is defined in terms of the protocol one uses to interact with it, and the behaviour expected in response to various protocol message exchanges, i.e. "service := protocol + behaviour"

[30]. The service definition also permits a variety of implementations. Example of services are FTP servers, networked CVS (Concurrent Version System) repositories, etc.

API: API stands for Application Programmer Interface – an interface (in terms of subroutine calls or object method invocation specifications) used by an application program for accessing services provided by some lower-level modules, for instance the operating system, JVM (Java Virtual Machine), standard C library, etc. An API may be language specific (i.e. defined in one or more programming languages like C,C++, Java) or be expressed in term of an Interface Definition Language (IDL), which is then automatically mapped onto a concrete programming language. It should be stressed that IDL is not a single language but a category of computer languages used to describe a software component's interface. Contrary to programming languages which are computer languages that can express all possible algorithms, IDLs deal only with a software component's interface specification.

The interoperability requirements highlighted in the previous section such as communication between distinct components and dynamic adoption of new components can be achieved by providing standard protocols, standard data exchange formats and standard APIs.

Protocols and exchange formats generally account for the exchange of data. APIs as they specify the behaviour and give access to the services, account for the exchange of algorithms. Interoperability can be in theory achieved by specifying only protocols or only APIs. However, relying only on standardised APIs requires different components to share the same implementation – the situation is not possible in practice. Protocols as they concentrate on external aspects of interoperability enable different implementations and are immune to implementation changes. On the other hand, standardised APIs, libraries, SDKs (Software Development Kits) are required to minimise development costs, enhance portability, and provide means

for the tight coupling required by high performance solutions. Thus protocols and APIs are equally needed to achieve interoperability [30].

3.2 Traditional techniques

The advantages of the modularisation of computer codes have been appreciated from the very beginning of the computing history. This appreciation has given the rise to various ways of decomposing programs into smaller pieces such as functions, objects, processes and components. Several techniques for the incorporation of those pieces into programs were introduced too. Due to their long history three particular techniques will be distinguished in this Chapter as the traditional ones: source code integration, software libraries and integration based on the interprocess communication.

3.2.1 Source code integration

The simplest technique for software integration is manual modification of the source code of two or more software pieces and making them interoperable. This accounts for instance to copy and paste from one source code to another, to reimplement the desired functionality or to write an interface code. This way of integration requires a thorough understanding of the integrated code, but may be tedious and error prone or lead to inflexible solutions [116].

An example of this approach is for instance Netlib [151] repository of FORTRAN source codes from which users can pick desired code and integrate it with their programs.

3.2.2 Software libraries

The use of software libraries is the most popular way to ensure code reuse and enable software integration. It mitigates the problems of tailor made source code integration

by introducing a well defined interface through which one piece of code can access the functionality of the other one. Additionally, by defining a set of standard libraries, independently developed applications can be made interoperable, providing that they use a common subset of standard libraries. However, this solution basically puts the problem of interoperability on shoulders of library writers, as the question of interoperability of libraries arises. Also, the term "set of standard libraries" is vague. Most of the applications areas have multiple representations in software libraries and the question is, who decides how the standard representation is defined.

3.2.3 Interprocess communication

Depending on which aspect one would like to stress, interprocess communication can be defined as the: *capability of an operating system that allows one process to communicate with another* or *a set of programming interfaces that allows a programmer to manage and coordinate different processes running concurrently in a single operating system or in many network connected systems.* [90]

Interprocess communication allows software integration on quite different level than the previous solutions, because the applications to be connected are treated generally as black boxes, and communication between them is done only through the public interfaces they provide. Except for the usually small portion of gluing code to fix incompatibility in protocols, no major programming work is necessary. Interprocess communication enables the building of truly modular solutions, however, depending on the application area the need to translate data between various "standard" protocols can be a bottleneck. Also in the case of many short-lived processes, the cost of creating and destroying them might be to high. There are various solutions for interprocess communication and they include: pipes and named pipes, message queueing, semaphores, shared memory and sockets [91].

3.3 Network enabled techniques

The techniques mentioned in the previous section (except for sockets) assume that connected components reside on the same machine. Nothing stops us however, from putting them on different machines connected by a network, and programmers have at their disposal many solutions based on network communication. They are based on an appropriate protocol and a client-server architecture. Because of the heterogeneity of computer networks, such solutions are built to be operating system and language independent.

Though at the first sight, network based techniques may look complex and handicapped by the large overhead of network related processing, it appears that likely network applications and Grid computing will be the software integration paradigm for the next few decades.

Below some of the most popular or most significant solutions for network based component integration are listed:[1]

CGI: Common Gateway Interface. CGI is an interface that allows the delivery to the user of dynamic web content by running through HTTP server special programs commonly called CGI scripts (though they can be implemented in any language). HTTP servers pass data to a CGI script either via environment variables, command line or standard output. Output data is passed back to the server by writing it to standard output. CGI requests are transferred via standard HTTP protocol.

RPC: Remote Procedure Call. This is a technique enabling to call a routine which is outside the address space of a given computer. In other words a local process can call a routine which resides on a remote machine by sending it arguments and obtaining the result. RPC is a whole infrastructure that separates programmers from the details of various operating systems and network interfaces

[1]This list is skewed with respect to operating systems, Windows specific solutions are not listed.

– programmers deal with just the function calls. Full scale implementations for RPC appeared in the late 1970s and the early 1980s but competing standards hampered RPC spread. Only when combined with the XML standard in the middle of the 1990s, in the form of solutions such as XML-RPC, RPC gained popularity again.

CORBA: Common Object Request Broker Architecture. CORBA is a standard managed by the Object Management Group (OMG) and the most brief description of this standard is that it is an "Object Oriented RPC". It consists of an Interface Definition Language (IDL), concrete language bindings and protocols that allow interoperation between applications.

SOAP: Simple Object Access Protocol. SOAP is a simple XML based protocol to let applications exchange information using HTTP protocol. SOAP specifies ways for invoking methods on servers, services, components and objects. SOAP specification also mandates an XML vocabulary that is used for representing method parameters, return values and exceptions.

3.4 Integration of multi-language applications

A separate issue is the integration of components written in different languages. If we neglect for a while network based techniques which can be also used for this purpose but with the cost of a network communication, then the integration of multi-language applications accounts for the generation of the so called gluing code which translates calls of one language API to calls of another language API. This translation might be direct or with the help of another, usually abstract[2] language generally called Interface Definition Language.

[2]An abstract language should be understood here as a language which does not have an interpreter or compiler.

3.4.1 Extending versus embedding

One can distinguish between two modes in which one language component can be linked with another language component:

Extending – adding new functionality to a scripting language by implementing it in another language and providing it usually via shared libraries. In extending the 'main' function comes from the interpreter of the extended language.

Embedding – calling a scripting language interpreter from an application not written in this language. The 'main' function comes from the calling application.

In practice these two approaches are often used together and they use the same API provided by the scripting language.

3.4.2 Classification of multi-language programming support tools

Multi-language programming can be done just by using appropriate APIs, but this way may require a lot of repetitious work, be error prone and time consuming. Usually the generation of codes for gluing multi-language applications is done automatically or semi-automatically on the basis of the signatures of exchanged functions or objects. Tools that provide such automation can be classified according to the number of languages they support:

"One to one" tools: Pyfort (`FORTRAN` and Python), f2c (`FORTRAN` and C), Boost.Python (C++ and Python), Sip (C++ and Python), f2py (`FORTRAN` and Python), mex (C and Matlab)

"One to many" tools: SWIG (C/C++ to many scripting languages)

"Many to many" tools: Babel [23], (also CORBA compilers, RPC generators, though they use a different paradigm).

3.5 Generic libraries

When developing software for scientific simulations there is a striking disproportion between time and effort spent on implementing the core number crunching part on one hand, and all the supporting code on the other hand. The high requirements for generality, flexibility, and portability make the situation even worse. That is, the support code (geometry handling, mesh generation, flexible data structures, visualisation) dominates the development and forces the computational scientist to deal with problems far from his area of expertise. Facing that, one usually has to "cut corners" what inevitably leads to the situation in which the code developed is very specific i.e. it is tailored to a particular problem and can not be easily reused in a different context.

One of the reasons for the code to be highly specific is tight coupling between underlying data structures and algorithms operating on them. It is possible however, using the generic programming approach [101], to separate data structures from algorithms. Generic programming deals with the generalisation of software components so that they can be easily reused in different contexts. Generic programming is mostly relevant, though not exclusively, to the languages with static type checking such as C, C++, `FORTRAN` , and Ada. This is the result of the static typing of variables that programs in these languages can be thoroughly optimised and translated into efficient machine code. However, static typing means also that functions can only operate on a strictly specified set of data types, even if other types offer the same functionality. In languages like C, it is possible to build generic solutions but the price for this is resigning from compiler support based on type checking. C++ in turn, offers solutions for generic programming while keeping strict type checking mechanism. These solutions are based on template programming and meta-programming.

3.5.1 Separation of data structures from algorithms

Probably the best known example of generic library is STL, the C++ Standard Template Library [99]. STL is a container class library, that is, it provides a set of container classes for modelling linear sequences such as vectors, lists, queues, etc. STL also provides generic versions of the linear sequence algorithms such as counting, sorting, partitioning, copying, etc. The containers are generic in the sense that they are independent of the type of items they hold and algorithms are generic in the sense that they are independent of the container type.

The STL library achieves its genericity by identifying a set of concepts fundamental to the domain of linear sequences. They include:

- *container* of some sequence of elements of arbitrary type,

- *iterator* providing access to sequence elements,

- *function object* which allows the operations undertaken on an object to be treated as data and pass that operation to algorithms,

- *algorithm* which work by applying an operation defined by a function object to a sequence given by a half-open ([`begin, end`)) interval of iterators.

The concepts are defined by specifying a minimal set of requirements which must be obeyed by user classes in order to work with STL. The concepts include for instance: STL Assignable – requires a model class to provide assignment semantics, STL Default Constructible, STL Equality Comparable and so on. A detailed description of STL concepts can be found in reference [104].

Among the domain specific generic libraries it is worth to note the availability of GrAL (Grids Algorithm Library)[13], CGAL (Computational Geometry Algorithms Library)[103], MTL (Matrix Template Library)[100].

3.6 Standardisation efforts

One very important aspect of software integration are various standardisation efforts. There are several government funded, industrial or the community based organisations undertaking the effort to provide common standards for languages, data exchange formats, APIs, protocols, etc. Without that effort any larger scale cooperation would be impossible. However, regardless of how much esteem we have for standards and how much we praise them, we should be aware that standards have the other, darker side. First of all people are using standards not just because standards are available but because they can profit from them. The second important thing to note is, that standardisation effort is very expensive and time consuming (sometimes just because of bureaucracy). This is why in order to be successful such effort must be usually funded by government or industry. The third point is that standards are often used in corporate wars. The company which controls a standard is automatically in a better position than its competitors. This is the reason why big companies are trying to persuade (or force) other to use their solutions as standards or are trying to sabotage unwanted standards just by not implementing some features or by adding their own extension and using their monopolistic position.

Despite this darker side of the issues, standards generally make our live easier. However good standards are seldom the effect of arbitrary decisions, more often they are the effect of an evolution of ideas and implementations. Also, not always the better idea wins, sometime an inferior solution takes precedence just because the implementation is accessible. This is why it is important to allow access to and quickly assemble implementations – this helps to evaluate and spread ideas. Cheaper ways to provide implementations are also an opportunity for less numerous and not so well organised communities to converge to their own standard solution. Here we specifically mean various groups of researchers working in the same field. The above was one of the driving forces for the research presented in subsequent chapters.

3.7 Conclusions

As can be seen from the above there are many techniques and even a more tools for each technique of software integration. The knowledge when and how to use a given technique is complex but essential for building practical, robust and scalable solutions.

The descriptions such as the ones provided in this Chapter can give researches a basic guidance to which solution to select. It is however necessary to realise that selecting the appropriate software integration mechanism is a complex issue influenced by many factors.

It should be stressed that not a single technique or a tool can be in a general case labelled as a universal or optimal one. However under the assumptions about the category of users presented in section 1.1 and requirements described in section 2.2 it seems that it is possible to indicate the techniques which are more likely to yield flexible and cost effective solutions. On the basis of the experience gained while experimenting with different tools, the techniques based on scripting languages will be advocated in the next Chapter as the most advantageous approach.

Chapter 4

Software integration based on scripting languages

This Chapter advocates scripting languages as a strategy for software integration.

Programming languages can be divided into two broad categories: system languages and scripting languages. While this is not a crisp categorisation, the distinction indicates that system languages are best suited for large software development from scratch while scripting languages are best suited for component gluing. This distinction and a brief characterisation of modern scripting languages is given in the first section of this Chapter. While probably best known with respect to administering operating systems and from CGI, scripting language applications go far beyond these, and they are used for almost any purpose. Scientific computing is not an exception and, in fact, computing environments based on scripting languages have a long history, for example Maple or MATLAB. The second section of this Chapter discusses the use of scripting languages for scientific computing, showing that they can be a valuable addition to traditional number crunching languages such as `FORTRAN` or C. The next section briefly compares general programming languages versus domain specific languages. General purpose scripting languages offer greater flexibility while domain specific languages may be easier to use for domain experts. The characteristic of some general scripting languages and their use for writing sci-

entific applications are briefly presented in the next section. Finally the last section discusses the efficiency issues which are commonly raised when scripting languages are mentioned in the context of scientific computing.

4.1 Characterisation of modern scripting languages

Instead of "scripting languages" the term "very high level languages" is more appropriate and should be used. "Scripting languages" might suggest that they are an inferior type of language and that "scripting" is not actually the same as "programming". This is of course not true and this false impression is partially created by advertising scripting languages as a perfect, easy solution, as the languages in which even one's grandma can program.[1] Scripting languages are fully fledged programming languages often providing more advanced features than traditional high level languages such as C/C++, FORTRAN 77/90[2]. They are often characterised by the following set of features:

- automatic memory management,
- dynamic typing,
- anonymous functions,
- closures,
- advanced string functions,

[1]This is yet another misconception because as author's experience as teacher suggests, students' problems with programming originate in difficulties to understand very fundamental concepts like control flow structures, variable references, local and global scopes, passing arguments by value, etc. Without understanding these features no syntax, and no standard libraries, however rich, are going to do much good.

[2]There might be a heated discussion concerning whether C is high level language but in spite of the new C99 standard it seems that it can be classified as such.

- built-in high level data structures such as lists, dictionaries, hash tables, etc., and

- extensive built-in standard libraries.

Another adjective which is used in connection with modern scripting languages such as Ruby or Python is "agile". Agile programming languages are characterised by being:

- excellent for beginners, yet superb for experts,

- highly scalable and hence suitable for large projects as well as small ones,

- suitable for rapid development,

- portable, cross-platform,

- embeddable, easily extensible,

- object-oriented,

- simple yet elegant,

- stable and mature, and

- endowed with powerful standard libraries.

It can be argued to what extent a particular language meets the above requirements, nevertheless this shows what programmers expect from their languages. The most popular scripting languages include: Ch, Guile, Matlab, Perl, Python, Rexx, Ruby, Scheme, and Tcl[3].

[3]Except for Scheme we skip in this list a whole category of function languages.

4.2 Scientific environments based on scripting languages

Scientific computing is commonly associated with the constant need for faster machines, larger storage devices and better algorithms. This is natural, as scientists need to simulate behaviour of more and more complex systems (multi-physics) on several resolution scales (micro and macro modelling). To fully tackle problems such as turbulent flow around an air-jet or climate modelling, there is still a lack of appropriate processing power, thus the stress on processing speed and efficiency. For other problems which can be effectively solved in a laboratory setup, resources are lacking to apply the solution techniques in real life, where crucial decisions have to be based on the results obtained. Should it be because of insufficient processing speed arising from the size of a problem or because of an insufficient processing speed arising from a combinatorial explosion of solutions caused by an inaccurate boundary condition or an inherent process nature (chaotic processes) – does not really matter. In the sense, that all these cases fuel the quest for faster processing. The quest for speed is also mirrored in the selection of programming languages where some languages are considered faster than other on the basis of execution benchmarks.

However it should be noted that, while important, the "Grand Challenges" and other cutting edge simulations are only a part of the whole area of scientific simulation research. Thus pure processing speed cannot be the single criterion for the selection of programming languages, programming techniques and computational technology. A program's processing speed is important for grand challenge problems or for industrial applications where even a small efficiency improvement multiplied by the number of installations times the number of runs gives a substantial gain.

However, it is suggested here that, for most of simulations run at academia research institutions the pure processing speed is not a primary issue. Such simulations are run mostly for insight and not for numbers to quote words of Hamming [133].

Thus not absolute processing speed but its relation to factors such as development speed, expressiveness, flexibility, learning curves, debugging facilities should be the criteria for selecting software solutions, and in particular programming language.

It should be also realised, that another important factor which shapes our view on software tools is the development of hardware and communication technologies. Though scientific communities are only at the beginning of the exploitation of Internet based technologies, things like Grid and mobile computing already shape our view on scientific computing. Successful projects such as SETI@HOME [117] or other similar projects have shown, that thanks to the global network, we are able to achieve unprecedented computing power. Sharing resources and wide collaborations are now important enablers in scientific computing. Scientific environments become more heterogeneous and spread. That of course, also shapes the software tools and among them the programming languages.

Some languages (notably C) are very efficient because they are close to the bare hardware. However when one introduces ideas like the "Net as a computer" and exposes the concepts of distributed and and parallel processing, then advantages of being close to "bare metal" diminish. It is apparent that other languages (e.g. Erlang [84], ZPL [156], Java) and among them also scripting languages can compensate processing speed by direct support of these new concepts.

The other factor mentioned – hardware development, also has a tremendous effect on the shape of software tools. Decreasing the cost of computing causes that in the majority of cases solutions which increase human efficiency in contact with computers are more cost effective, than solutions that increase the efficiency of computer programs.

The whole discussion above is to refute the most often raised objection against scripting languages in scientific computing which is their efficiency. They are less efficient in processing speed but their other features make them an important complement to traditional "number crunching" languages.

The value of scripting environments for scientific simulations has for a long time been recognised by scientific communities. The first, `FORTRAN` implementation of Matlab, probably the best known environment for numerical simulations, was designed in the late 1970s. Matlab was rewritten in C in 1984 and at that time The Math-Works company was founded, which continues Matlab development to this time. In the GNU/Linux environment there is a well known package Octave, which is sometimes called "Matlab clone"[4]. Octave was created in the early 1990s and is still actively developed. The French Open Source Scilab [118] belongs to the same class of environments. Most of the popular scripting languages provide specialised libraries for numerical computing. Thanks to them the language interpreter can turn into versatile tool for creating simulation codes. In Python the most interesting are the `Numeric` package and its newer version `Numpy` aimed at providing efficient tools for matrix manipulation in the spirit of Matlab, and the `SciPy` package being an impressive collection of various numerical libraries listed in Table 4.1. There is also a growing popularity of the Ch programming language [93, 92]. Ch is an interpreted superset of C and a subset of C++ providing specialised numerical libraries. Actively developed, Ch can become a close competitor to Matlab. Chapter 10 provides detailed description of extending the Ch environment with new simulation tools.

4.3 Hybrid systems as a model for academic simulation systems

4.3.1 What is a hybrid system

For the purpose of this thesis a *hybrid system* is defined as software package consisting of four basic components:

A – specialised libraries written in system programming languages, compiled for

[4]However, as the main Octave author John W. Eaton confesses, it was never thought to be such [26]

sparse	Some sparse matrix support, factorisation, solving
linalg	linear algebra (ATLAS + LAPACK)
cluster	information theory functions (currently, vq and kmeans)
weave	compilation of numeric expressions to C++
cow	parallel programming via a Cluster Of Workstation
fft	fast Fourier transform module (fftpack and fftw)
integrate	numeric integration and ODE solvers
interpolate	interpolation of values from a sample data set
optimise	constrained, unconstrained, root-finding algorithm
signal	signal processing
special	special function types (bessel, gamma, airy, etc.)
stats	statistical functions (stdev, var, mean, etc.)
ga	genetic algorithms

Table 4.1: Modules of SciPy – scientific package for Python.

efficiency,

B – scripting programming language (or languages) providing an interpreted environment

C – interface modules providing a bridge between specialised libraries and the scripting language environment,

D – tools for creating new interface modules for the user's custom codes.

Components A and B do not require much further comment. Component C is considered separately as the interface modules can be provided with specialised libraries in A or can be built by independent vendors. It is important to provide ready interfaces and not just the components A, B and D because building scripted interfaces to some libraries can be quite intricate and beyond resources of most users. It is very important to consider the component D as an integral part of the hybrid system. Without D users would be not able to integrate their custom codes into the scripting environment and would be restricted in the possible customisations of the existing code.

Depending on the characteristics of the above components, it is possible to build various flavours of hybrid systems. Table 4.2 shows a snapshot of the possible combinations. The cases 1, 2, 3, 4, 5, 6, 8, 9 use, according to the description given in section 3.4, one-to-one type of integration tools. Cases 7, 10 and 11 are examples of the usage of one-to-many integration tool (SWIG) which can produce C/C++ wrappers for various scripting languages. Case 12 is an example of the many-to-many integration approach. Of course, the table presents only aspects **B** and **D** of hybrid systems.

4.3.2 Exemplar hybrid system

It is not possible to point to the combination of components **A, B, C, D** which would yield "the best" hybrid system. First of all, it is not clear by which criteria

	System language	Scripting language	Integration tool
1	C, Fortran77	Matlab	mex
2	C++	Matlab	matwrap
3	C, C++, Fortran77	Octave	mkoctfile
4	C++,	Octave	matwrap
5	Fortran77, Fortran90	Python	f2py
6	Fortran77, Fortran90	Python	pyfort
7	C, C++	Python	swig
8	C, C++	Python	sip
9	C, C++	Python	Boost.Py
10	C, C++	Guile	swig
11	C, C++	Tcl/Tk	swig
12	C, C++, Fortran77/90, Java	Python	babel

Table 4.2: Samples of possible hybrid environment configurations.

such a system should be judged. Secondly, the optimal choice strongly depends on the application area. Finally, similarly to picking up a programming language, the choice depends also on non-technical factors (like previous exposition to particular programming languages, the language used by other members of research team, etc).

Stated this, the author presents his selection for a hybrid system for building finite element (FE) based simulation tools:

A – system programming language: C/C++.

In building FE systems the author is more interested in pre- and postprocessing modules and it seems that C/C++ offers the best choice of visualisation, geometry and data structure libraries. Besides, C++ has a very good support for generic programming, and generic programming is claimed in this thesis to be one of the most important foundations for building flexible systems.

B – scripting language: Python.

This choice is dictated by: a very clean syntax, rich standard library and many scientific tools, combined with a very good support for integration with C/C++.

C – interface modules: here the choice depends on specialised libraries in question. Usually the choice is restricted as there is in most cases a single interface implementation to the given library.

D – integration tool: SWIG.

It is stated in [7] that SWIG (Simplified Wrapper and Interface Generator) is "an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby and Tcl". The world "simplified" might be actually a little misleading. While the basic usage of SWIG is simple, the program is in fact a really advanced, customisable compiler, handling the whole C and almost all the C++ syntax. The SWIG project was started in 1995 by David Beazley, and since then it has became the *de facto* standard in the domain of interface generators for C/C++. During the years since 1995 several new language modules were added, and now SWIG supports: Guile Java, Mzscheme, OCALM, Perl, PHP, Python, Ruby, Tcl, Chicken, C# and XML. Chapter 10 describes the development of a new module for Ch language. Table 4.3 gives SWIG characteristics in terms of supported C/C++ features.

4.4 General versus special purpose languages

The previous section has shown several examples of using scripting languages to build scientific software. Naturally, the question arises, what kind of scripting language is best suited for scientific applications. Here we will consider two broad

C
Handling of all ANSI C data types
Global functions, global variables, and constants
Structures and unions
Pointers
Arrays and multidimensional arrays
Pointers to functions
Variable length arguments
Typedef
C++
Inheritance and multiple inheritance
Overloaded functions and methods
Overloaded operators
Namespaces
Templates as template members
Template specialisation and partial specialisation
Smart pointers
Library support for strings, STL vectors, and more

Table 4.3: C/C++ features supported by SWIG [7].

categories of scripting languages: general purpose languages and domain specific languages.

General purpose languages are understood as languages not tightly coupled with any particular area of application, while domain specific languages are the ones, which are designed with a concrete application domain in mind.

Programming languages such as Python, Tcl/Tk and Scheme are examples of general purpose languages, while Octave, Matlab, FreeFEM and AWK can be classified as domain specific languages.

The choice between a domain specific and a general purpose language appears naturally not only with connection to scientific environments but in the design of any software system that requires a programmable interface. The designers of such systems have basically three choices, or trade-offs, which will be elaborated shortly:

- to adopt an existing language as a programmable interface,
- to design and implement a new language completely from scratch,
- to provide a new language but built as an extension or restriction of an existing one.

Using an existing language

This choice has the advantage that no design and almost no coding effort is necessary. Also by grabbing an existing language, users of the programmable interface do not have to learn any new syntax. The main disadvantage of this solution is that users will be forced to put some effort into expressing application abstractions in a general purpose language syntax.

Writing a new language from scratch

By deciding to build a completely new language the programmer frees himself from restriction of the syntax of an existing language and from the restrictions imposed by the language compiler or interpreter. In this case the syntax of the designed language

can closely resemble the notation used in an application area where the language will be used. Thus there will be no or a quite flat learning curve for users, and users will quickly decipher written programs. However, these advantages are mitigated by the fact that, despite enormous progress in compiler technology, designing and implementing a programming language requires expertise and considerable effort. One has to put effort not only into the immediate coding but also in maintaining the language – fixing bugs, writing documentation, including possible tools such as profilers or debuggers, etc.

Extending/restricting an existing language

The third choice is a compromise between the last two. Here, a programmable interface is provided by restricting or extending functionality of an existing language. While still being restricted by the syntax of the host language, this solution allows the development costs to be cut considerably. Also, the reliance on an existing language gives better access to programming tools, libraries, and a greater possibility to extend the programmable interface with third party software. Scripting languages are especially amenable to such tweaking, taking into account their dynamic nature, visible in the facilities for introspection or for assembling functions, or classes on the fly.

4.4.1 Does the programming language matter?

One of the very popular topics for discussion among programmers are the advantages and disadvantages of particular languages. Sometime one can hear a more general question, about whether the choice of a language matters at all. Heated discussions that might be invoked by this question are partially a result of different understanding of the term "language" it the above question. The term "language" can be understood as a set of syntactic rules, a set of underlying semantic concepts (like objects for C++, processes for Erlang, or stacks for Forth) or finally, as a pro-

gramming environment (standard libraries, compiler and other development tools). Based on his experience,[5] the author is in the position that for most of the classical languages the syntax does not matter. Even if the syntax is a little unusual like for Tcl, or more exotic like the reverse Polish notation of PostScript, experienced programmers can quickly accommodate this. Much more important is the set of underlying semantic concepts. If these concepts match the entities and processes modelled with a given language, then the language will act as a natural notation for relations in the domain modelled. This naturally helps to avoid gross design mistakes which can lead to project failures. For instance, when building a highly distributed concurrent system it would be wiser to chose Erlang than C, because of Erlang's notion of processes built into the language. Finally, the third aspect – programming environment – directly affects programmers productivity. Given the right set of tools, the programmer can concentrate on more important architectural issues than the implementation of low level utilities. A good set of standard libraries can sometime compensate for the mistakes committed when selecting a language with respect to the underlying semantic concepts, though one can not always count on that.

From the above discussion an important conclusion emerges, that, in real world situations, none of the known languages will completely dominate. In the last twenty years it was claimed that C++, then Java, then C# were the silver bullets for programming problems. Nothing like that came true. The phrases like "On Language for All"^{TM} under which Ch is advertised, are catchy marketing tricks. It is true that Ch can sometimes play the role of `FORTRAN`, C, Matlab or Java, and be used for numerical computing as well as for web application development, but that does not make it universal. Everything depends on what kind of a real system the language has to describe.

---

[5]The author teaches introductory programming in C, object oriented techniques in C++ and numerical methods in Octave. On an everyday basis he programs in Python, Tcl, AWK and from time to time deals with legacy Fortran 77 codes.

Thus, in author's opinion, we have to accept a world with many programming languages and what is more important, as the interactions between various scientific fields and application domains become more intricate, we will more often encounter the necessity to use more than one language in a single project. This is why enabling seamless interaction between multi-language components is an important issue.

4.5 Efficiency considerations

Programs' execution speed is often an opiate for programmers and sometimes when mentioning a scripting language others comment on how slow these must be. This is of course a fallacy, as one has to distinguish between program execution speed and its development speed. It is true, that because of being interpreted, scripting languages are by definition slower than their compiled counterparts. However, the size of this gap depends on a particular application, and it is also quite interesting to look at absolute difference in execution time.

Table 4.5 taken from [8] shows execution time for three versions of a finite volume based program for modelling phase change effects. The first program is a plain Python version, the second one automatically translates at run-time some of the most critical sections to C code, and the third version is coded in `FORTRAN`. Comparing the execution times one can notice that the plain Python version is around 25 times slower than the `FORTRAN` version, what can be reduced to around 10 times if the `--inline` directive is used. Does the Python versions perform very badly? Not really – in the absolute time it took the plain Python version about 2 minutes to calculate the result for the biggest mesh, compared with 4 seconds in the case of the `FORTRAN` version. Now, the real question is how much time can be saved when designing and coding this program in Python instead of `FORTRAN`? A simple calculation demonstrates, that if we save just two hours on program development, then until the program is run more than 30 times, the Python version saves more time.

Elements	FiPy [s]	FiPy –inline [s]	FORTRAN [s]	FiPy memory [kB]
100	0.29	0.31	0.05	30068
400	0.45	0.24	0.04	31260
1600	1.18	0.60	0.06	34280
6400	5.47	2.02	0.19	47864
25600	27.2	9.63	1.01	91872
102400	115	42.37	4.17	269332

Table 4.4: Comparison of raw CPU time and memory usage for the FiPy used to model grain growth and subsequent impingement [8].

Thus if considering execution speeds at all, one has to take into account how many times a program will be run and how much time can be spared in the development process. Of course this is not a justification to make sloppy, inefficient implementations when we can do better. However, if we are building prototypes, throw-away programs, or programs that we know they will not be used many times then shortening the development time is more important than shortening the execution time.

4.6 Concluding remarks

This Chapter advocated the use of scripting languages for the integration of software components. After giving an overview of the modern scripting languages, their use for scientific simulation codes was discussed. For the purpose of this dissertation the notion of a hybrid system was introduced in section 4.3. The examples of various configurations of the hybrid systems were given, and on the basis of the experience gained while experimenting with these configurations, the configuration based on Python as the scripting language and SWIG as the wrapper generator was selected as the one giving the most flexibility with minimal overhead.

This Chapter discussed also the selection of the programming language for scientific simulations. This discussion has shown that in modern simulation codes it is likely to encounter the necessity to handle interactions between components written in different languages.

The last issue discussed in this Chapter was the issue of balancing program versus programmer performance. It was shown that, while there is a performance loss when comparing execution time of a program written in a scripting language to the execution time of a program written in a system language, the overall performance of a scripting language program can be quite acceptable especially for prototype or educational programs. Additionally, the drop in the program performance can be effectively counterbalanced by the increase of programmer productivity.

The next Chapter will provide a detailed discussion of the first component of hybrid system, namely the specialised libraries written in a system programming language. It will be shown that in order to match the characteristics of the complex simulation systems presented in section 2.2, these specialised libraries should be based on the generic programming paradigm. It will be shown how to apply the generic libraries to build a mechanism for linking the components based on the geometric models and grid abstractions.

Chapter 5

Grid and Geometry Exchange Services

The two previous Chapters discussed the need for developing widely acceptable standards for the exchange of simulation data and the difficulties in achieving such a goal. As a continuation of that discussion this Chapter gives a short overview of the desirable properties of a flexible simulation environment and points to the UNIX-like environment model as a practical solution for data exchange and component integration problems. Such environment is characterised by the domination of pipeline processing models, a rich set of generic tools and a flexible mechanism for connecting them.

The case of exchanging geometric and grid based data is discussed as the example, where such processing model can be successfully applied. Then concepts of "geometry bus" and "grid bus" are introduced as concepts which will enable effective exchange of such data. Finally the development of an original idea of Grid and Geometry Exchange Services (GAGES) is described. This idea will unify several concepts presented in the previous chapters and it is the basis for providing an effective component integration mechanism for the scientific simulation codes.

5.1 Monolithic versus modular applications

There are two basic trends in building and using computer applications. One is to build monolithic applications which offer users the primary as well as all sorts of secondary processing facilities. These applications are monolithic in the sense, that they are programmed, installed and used as a single entity. The second trend is to build modular applications. In such applications the primary data processing task is done by the main program unit, but most of the other, non crucial processing tasks, are left to external units. We can thus distinguish primary and secondary program modules. The secondary modules do not have to be present in order to use the primary ones. Monolithic applications can also be built from modules, but the mechanism of the module interaction is hidden from the users. In modular applications in turn, the interaction mechanisms are purposely exposed and made user friendly. Of course, the above classification is not always sharp, nevertheless visible. What is more, such a distinction can be projected onto the way people use applications and what they expect from the software.

5.2 Manipulation of geometric and grid based data

Data manipulated during pre- and post-processing stages of a simulation run can be roughly categorised as geometric and grid data. Geometric data describes some continuous regions in space together with properties assigned to such regions. Grid data in turn, comes from the process of discretising geometric data and properties defined with respect to geometries, such as a mass density or temperature distribution. The grid is a very important concept upon which several computational algorithms are based. A more detailed descriptions of the concepts relating to geometries and grids are given in Chapters 6 and 7, respectively.

Geometries, though this is not a strict rule, are manipulated most of the time by monolithic, usually commercial, CAD programs. The manipulation accounts

for creating points, edges, surfaces, volumes, applying geometric transformations to them, assigning physical properties and visualising them. Creating a geometric model of an object is the usual prerequisite for generating a discrete grid over it. Processing grids appears to be a little easier and there are more tools for processing them, than there are for geometries. Grid processing tools can be equally well monolithic as modular. There might be stand alone tools for grid generation, optimisation, quality measurements, partitioning, interpolation of grid data, geometric transformations and visualisation, as well as dedicated all-in-one applications.

Processing geometric and grid based data brings some problems from the point of view of building flexible simulation systems. The need for standardisation of geometric descriptions has been early recognised by industry, especially aerospace and automobile industries. As a result we now have a couple of strong industrial standards for geometric descriptions: STEP, IGES, DXF, ACIS SAT, etc. There is also an abundance of CAD programs supporting these standards and providing excellent tools for geometry manipulations. However, the industrial standards are usually too complex for the purpose of academic research. Implementation of simple and medium complexity tools based on these standards is usually not cost effective. There are some proposals for a new geometric description standard for use in academia [18, 9] but even when it becomes established, we are facing a long period of preparing freely available tools based on this standard.

In the case of processing grid based data the situation is different but also far being from ideal. First of all, there is nothing like a common standard for exchanging and manipulating grid based data. There are several standardisation proposals and tools used in different fields: CDF, netCDF, HDF, HDF5, and the formats of each the bigger bigger suppliers of simulation systems. Based on the popularity of the XML language there are also some XML based proposals for scientific data description languages [53, 54]. This situation is caused by the very complex nature of the grid concept which is discussed in detail in section 7.2.

However, even if we take into account a narrower field like finite element modelling (FEM), there is no widely accepted standard for the FEM data. There are some standardisation proposals [55] and maybe one day the format of a big FEM software vendor will prevail, but at the moment each research group implementing FEM algorithms usually uses its own home grown format. Similarly, most of the visualisation packages besides supporting most popular formats, introduce their own ones. So, though we may find several useful tools, using them in a single tool chain often requires the effort of translating between their data formats. In the case of ensuring interoperability on an API level the situation is even worse. There is nothing like a common API for grid processing. Of course again, due to the complexity of the subject it is hard to expect a truly universal API, but we are not even close to some partial solutions.

5.3 Geometry bus and grid bus

For the reasons explained in section 3.6 we should not expect that soon we will converge to some common APIs of data formats for grids and geometry. Thus, if one wants to build flexible and modular simulation environments, one has to find the way to ensure interoperability in a very heterogeneous world of grid and geometry descriptions, and what is more, that solution must be at a reasonable price.

That brings us to the "geometry bus" and "grid bus" concepts. These concepts are best illustrated in Figures 5.1 and 5.2. Trying to define these concepts we can say that:

The geometry bus – is a way of enabling interoperability of several programs which use a geometric description as a basic entity upon which they operate on.

The grid bus – is like geometry bus but this time the basic entity that is manipulated is grid based data.

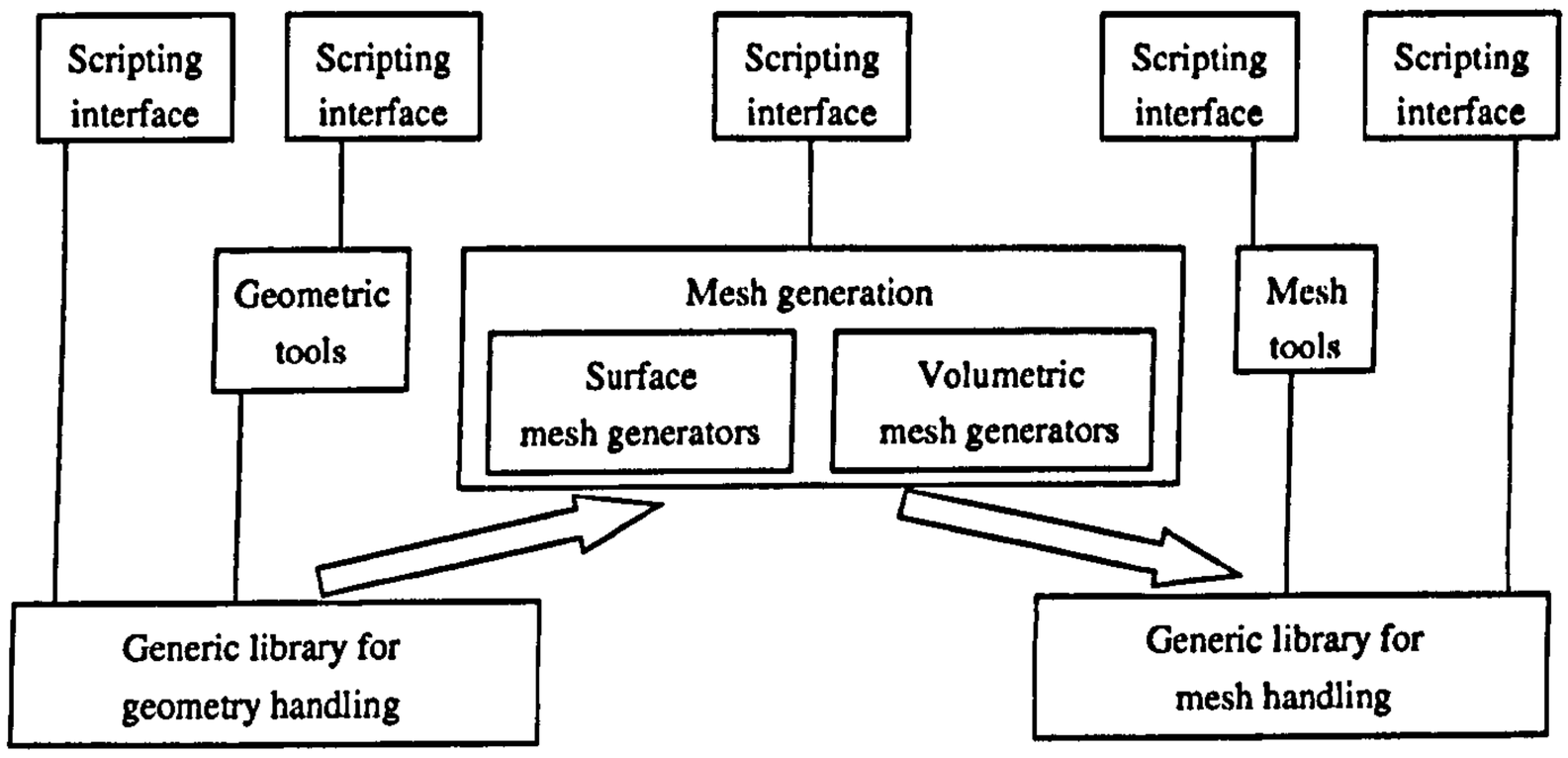


Figure 5.1: The structure of GAGES framework.

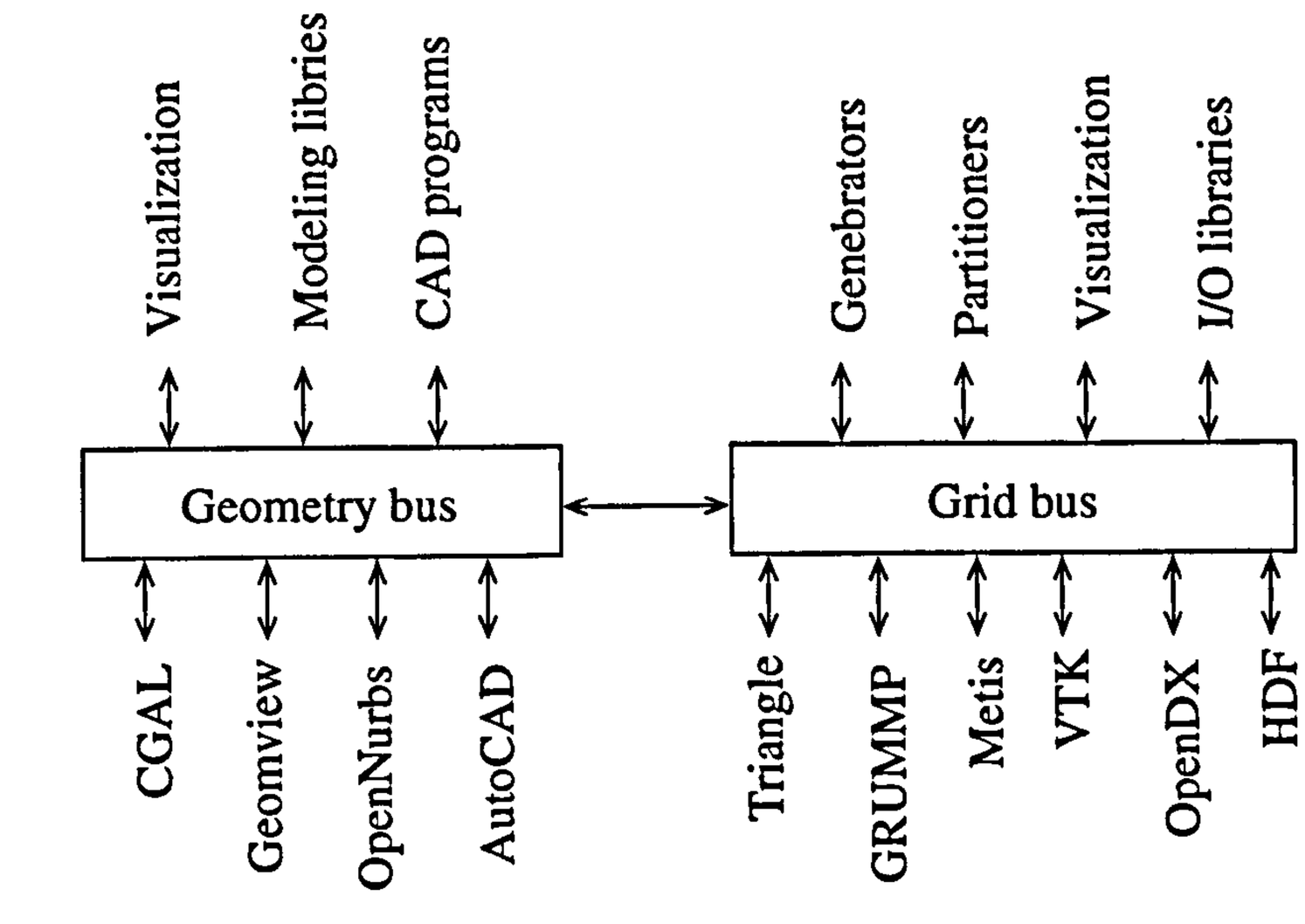


Figure 5.2: "Geometry bus" and "grid bus" concepts.

Looking at the above definitions the concepts seem quite similar, thus in further discussion we will refer only to the grid bus, making the necessary comments if the solution for the geometry bus differs considerably from the one for the grids.

Assuming the following scenario is imagined: a user takes polygonal description of some geometric domain, generates an unstructured triangular mesh from it, partitions the mesh for parallel processing, then uses the mesh as input data to an analysis program producing as a result another mesh with attached nodal data, and finally visualizes the results.

In this, we can have four separate programs: mesh generator, mesh partitioner, analysis program and visualisation program. In the light of the grid bus concept it should be possible to easily build a processing pipeline from these programs by connecting the output of one to the input of another (e.g. by standard a UNIX pipe mechanism), or if the programs are provided as libraries, it should be easy to create a "master routine" governing the execution of the processing units.

A better illustration of the grid bus concept can be given if we take a more detailed example of an algorithm involving iteration over boundary faces and face vertices. Such iterations may be required for instance when calculating object surface area, nodal load from the hydrostatic pressure, or checking for collisions with other objects. The iteration over boundary faces is always possible for the topologically valid mesh, regardless how the mesh is represented. However in some representations, for instance "reduced interior representation" discussed in [4] the boundary mesh is stored explicitly, thus iteration over it is a trivial task. Other representations, in turn, may require more work to be able to enumerate boundary faces. For instance, in the most common finite element mesh representation one stores for each cell references to its nodes. Thus, in order to iterate over boundary faces, one has to find cell to cell adjacency relation, and then using the generic cell description check on which cell face there is no neighbour. Such face is the boundary face. The above calculations are possible, regardless of how the mesh is represented.

The only requirement is that the representation allows the iteration over cells and over cell vertices.

Transformation of the iterations over cells and cell vertices into the iteration over boundary faces is an example of what happens in the grid bus. But there is more to that. In order to plug a concrete data structure into a grid bus one has to write a data structure adapter. If through this adapter grid bus sees that the data structure provides direct iteration over boundary faces, then this direct way is used instead of the generic one. In case of C++ implementations this can be achieved leveraging generic programming techniques. Also on implementation level it should not matter if the iteration is provided in terms of integer indices, pointers or iterator objects. Grid bus provides mechanisms which allow to compensate the differences in data structures representation, providing that the data structures represents the same mathematical concepts. Of course, the price to pay for the use of the generic mechanism is the software efficiency, thus grid bus should be constructed in a such way as to allow trade-off between solution genericity and efficiency. This can be done for instance using C++ generic programming based on templates, where by template specialisation one can provide a more specific and efficient solutions. On a coarser level it can be done by turning a grid bus into a hybrid system. In this case connecting components to a grid bus can be done on a more generic scripting level or on a more efficient compiled language level.

A simplified view on grid bus could be to treat it as a representation independent API for grid manipulations. This is correct, however from the practical point of view the API specification is not enough. First of all, writing such API solely from the theoretical considerations is hard. Even if such API could be designed, it has to be backed by a sample implementation anyway.

Designing the API and then implementing libraries for it would be a top-bottom approach. What this thesis proposes is a bottom-up approach: selecting a base library which can be used as a common denominator for many representations,

augmenting it with mechanisms for efficient components linking (for instance by turning it into a hybrid system), and then eventually formulate an abstract API. In order for this schema to be feasible the base library should be quite versatile and thus the attention was directed towards generic programming approach as explained in section 3.5. On a next level, different base libraries can be linked themselves, thus creating a hierarchical system of components connections.

It should be also stressed that the component linking mechanism expressed by the grid bus is a static one – that is, a piece of the software connecting a given component and the grid bus must be written beforehand by a user. Except for some low level mechanisms offered by C++ template system, there is no automatic discovery, checking, and matching of components' interfaces. Thus it is not possible to just specify a data source component and a data receiver component and then to allow the grid bus to autonomously resolve how to connect these components. Such level of "intelligence" would be very desirable, for instance in the systems based on the blackboard architecture, but it is not considered in this thesis. Nevertheless, the solutions presented in this thesis can be a good starting point when constructing such dynamic, "intelligent" component linking mechanism.

Well, the goal of geometry bus looks very, very ambitious. It is of course impossible to ensure interoperability between any two programs or libraries. The goal of grid bus is not to provide the single correct solution and world domination. Its goal is rather to identify subsets of tools showing common roots and only when these subsets are identified and generalised, then trying to devise connections for such tools. The second goal of the grid bus is to propose a practical way of achieving interoperability, practical in the sense that it will yield working solutions in a reasonable time and at acceptable costs.

5.4 Grid and Geometry Exchange Services

This section presents the the main idea of this thesis, that is, the idea of Grid and Geometry Exchange Services (GAGES). GAGES introduces a new strategy for the integration of software components and on the practical side it is a concrete example of building a hybrid system in the domain of the grid and geometric models manipulation.

Geometry bus and grid bus, together with a mechanism connecting them, is what constitutes GAGES. This is symbolically depicted in Figure 5.3.

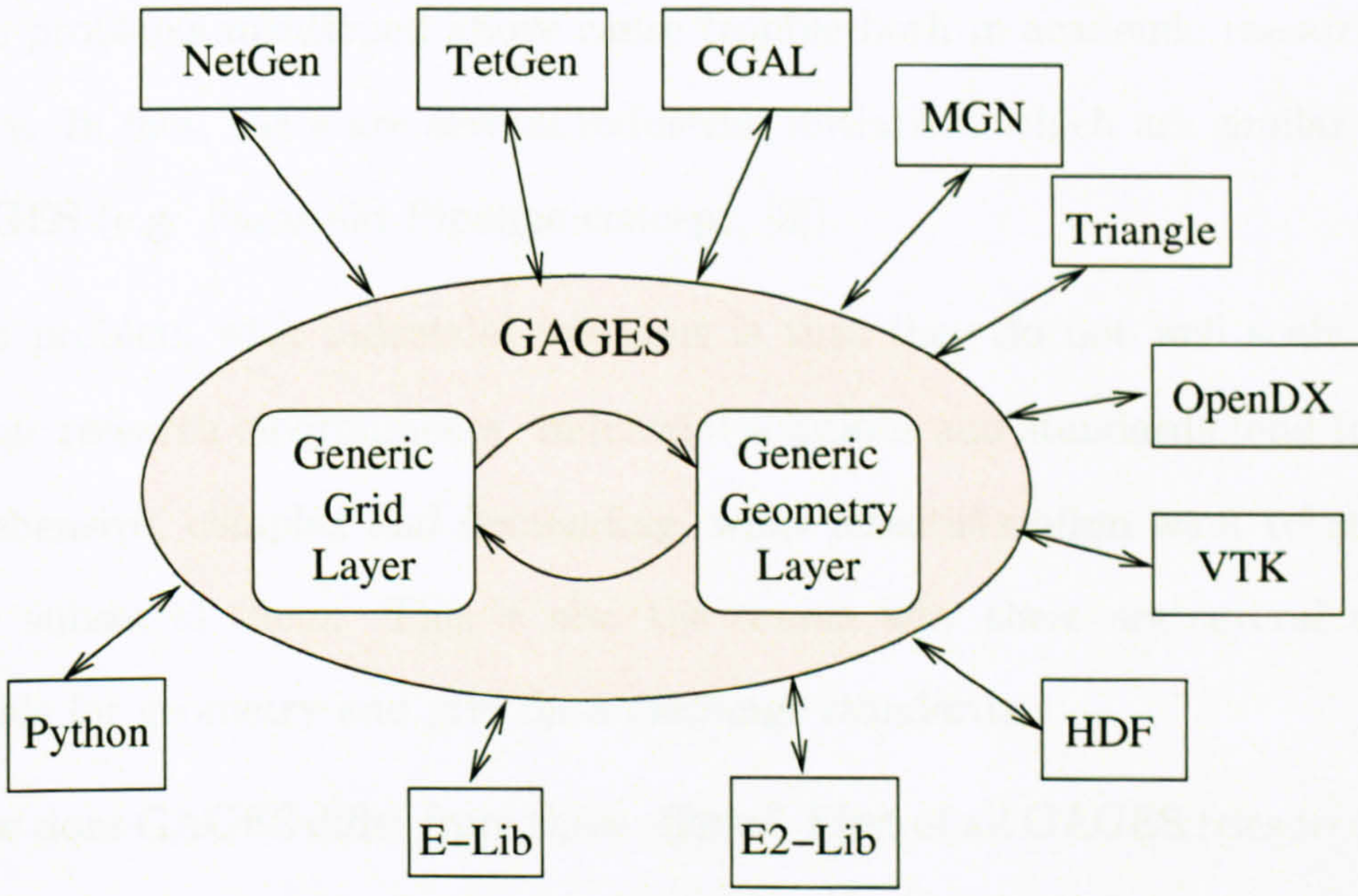


Figure 5.3: Conceptual view of GAGES.

GAGES is to be thought of as ways of allowing researchers to freely and reliably exchange their digital models and tools for processing them, especially the parts which concern the geometry and grid. The role of GAGES is to support and promote algorithms and data exchange by providing a set of generic services. GAGES is geared towards small research groups or even single researchers working in academia (which of course does not preclude other users).

A common way to reuse algorithms is to encapsulate them into software libraries. It is roughly speaking similar to using off the shelf components like nuts, bolts,

springs, etc. in mechanical engineering. However, the mechanical components are highly standardized and are meant to fit together, while scientific libraries are not so easily used together.

Similarly, sharing data can be accomplished by using some common data format. Yet, there are not widely accepted standards (especially in the case of gird data exchange) or there are several competing complex standards (espccially in case of geometry data exchange). The other problem with data exchange standards is that there should be many of tools supporting the standard which often is not the case.

The problems mentioned above cause trouble both in academic research and in industry. In fact, there are several industrial initiatives which are similar in scope to GAGES (e.g. Parasolid Pipeline concept, [3]).

The problem with industrial solutions is that they do not well scale down to academic research environments. Industrial solutions and standards tend to be very comprehensive, complex and demanding, while scientists often want to use only a narrow subset of them. This is also the reason why there are several academic proposals for geometry and grid data exchange standards.

How does GAGES differ from these efforts? First of all GAGES tries to overcome the biggest shortcoming of most of the academic efforts which is the lack of tools supporting the proposed standards. The dissemination of any standard is strongly hampered by the lack of tools supporting it. Secondly, GAGES focuses more on reuse of algorithms, i.e. on connecting different libraries and software packages, than on defining yet another data exchange format. We have to accept the situation where several formats exist, but if the tools supporting those formats can be brought to work together, then the formats will become more interchangeable. The third property of GAGES is that it should be possible to implement its architecture using already existing software. GAGES is being developed from a very utilitarian point of view, assuming that due to the complexity of subject that the good solution can be obtained only by implementation, experimentation and refinement cycle, contrary

to the pure juggling of abstractions.

To reach its goals GAGES leverages a generic programming approach. The main idea is to select a small number of highly abstract, generic libraries for geometry and grid handling and to connect them to several other more specialised tools for geometry modelling, grid generation, domain decomposition, data I/O, scripting, etc. Those generic libraries will provide the necessary high level data structures and algorithms to which more specific data structures and algorithms will be mapped. This way one may hope to obtain very flexible and open environment at low cost. If process of mapping concrete software components to the generic ones can be made simple or even automated, then scientists will likely add their own tools to it, providing mutual benefits to them and to the whole users community.

5.4.1 Layered structure of GAGES

Another important feature of GAGES is its layered structure. In GAGES four layers can be distinguished and they are shown in Figure 5.4. `GrAL` and other core libraries

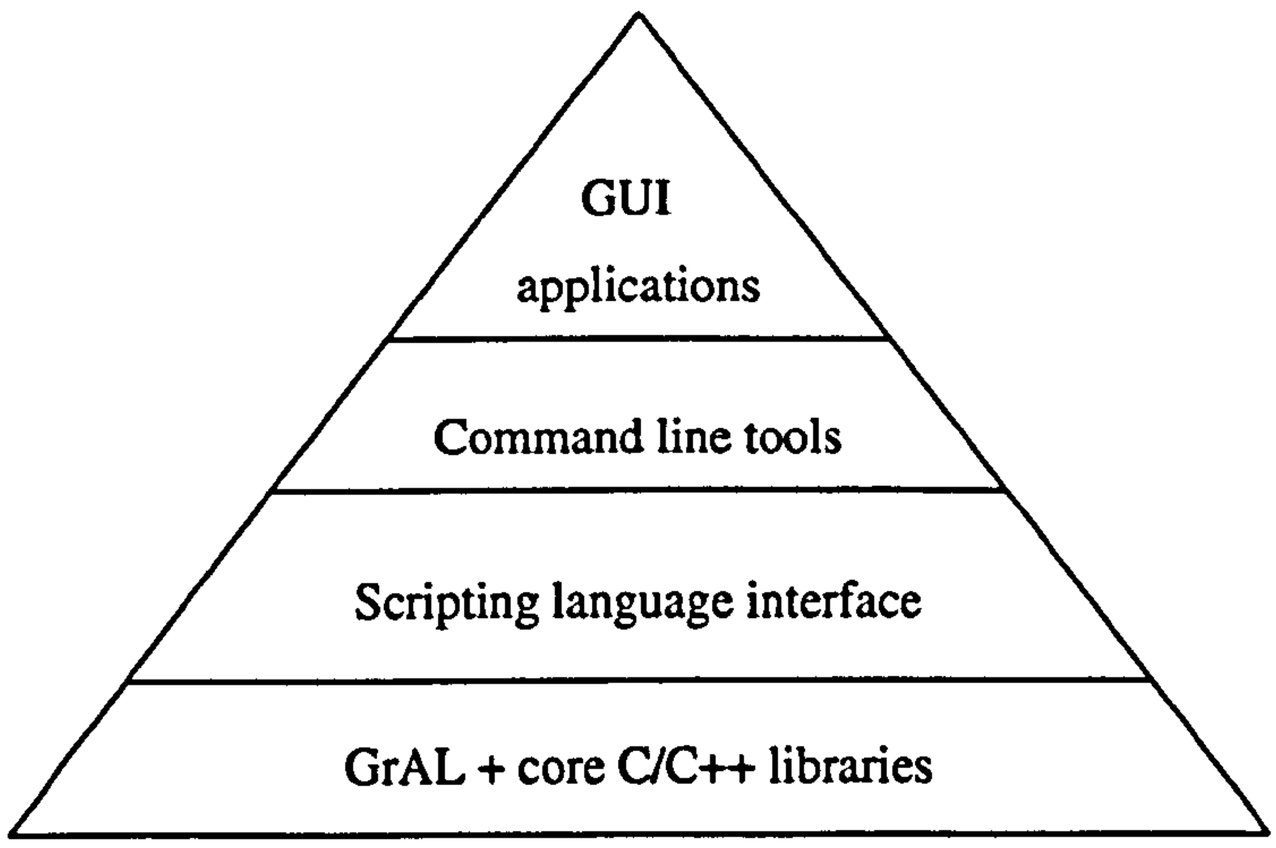


Figure 5.4: Layered structure of GAGES.

form the foundation on which scripting language packages are based. One of the key assumptions of GAGES is that all component integration which is possible using compiled components should be also possible using scriptable components. This

duality, though quite demanding in terms of development effort, offers the great possibility of balancing development speed and interface flexibility versus run time performance. The third layer from the bottom is the layer of command line tools. Components in this layer will be most often built using scripting language programming. Elements of this layer will follow the philosophy of UNIX tools. They will offer the possibility of coarse grained component integration based on interprocess communication facilities. The extent to which this layer will be utilised will be partially dependent on support from the operating system environment. Finally the top layer is the layer of the GUI based components. In terms of automatic component integration this layer is less interesting because the GUIs assume almost exclusively by an interactive usage. However in some cases, this layer will be indispensable, for instance in data discovery in virtual environments.[1]

5.5 Research goals

Chapter 3 sketched the background for the need of the software integration and at the beginning of this Chapter such need was shown on the example of the pre- and post-processing of scientific simulations. The idea of GAGES was introduced as a new strategy for developing practical solutions to the problem of the exchange of geometry and grid based data.

Sketching such strategy is however only one side of the coin. The other one, much more difficult, is to prove that this strategy can be used in practice. The main difficulty lies in the selection of the technical solutions, and the verification to which extent the selected tools and techniques fullfil the assumptions. Practically it means that several software components must be installed, their documentation or source code analysed, interface modules for their integration must be implemented and their working should be tested on some examples. The conclusions driven from

[1]The GUI layer should be distinguished from the concept of graphical output. The distinction lies in assumption of real time interaction with an application in the case of a GUI.

the observation of this process, plus solutions of the concrete software engineering problems, are what constitutes the main contribution of this thesis. The above is a complex task, thus it was divided into several research goals. The most important of these research goals are listed below with references to the respecive chapters or sections describing them in details:

1. Investigation of the geometry bus related issues. This accounts for:

   (a) Overview of geometric models (section 6.1).

   (b) Investigation of geometric model exchange standards (section 6.2)

   (c) Selection of the base library for the geometry bus (section 6.4)

   (d) Implementation of the missing functionality for linking geometry and grid buses (section 6.4)

2. Investigation of the grid bus issues. This involves:

   (a) Investigation of data structures for representing grids (section 7.2)

   (b) Selection of the base library for the universal grid exchange layer (section 7.5)

   (c) Implementation of the links between the base library and the utility tools: mesh generators, visualisation tools and libraries, data format converters, etc. (section 7.6).

   (d) Design and implementation of a scripting interface to grid based components (Chapter 8).

3. Implementation of a surface mesh generator as the necessary link between geometric model and its grid discretization (Chapter 9).

4. Analysis of interface generation tools for the multi-language programming (Chapter 10).

5. Development of a Web interface to the GAGES services as an important case study of application of a hybrid system (Chapter 11).

5.6 Potential benefits from GAGES

GAGES can bring very real benefits to the academic research community. First of all it will enable or make easier the usage of a variety of tools. Services offered by GAGES will spare users the effort of implementing their own adapters between popular tools. GAGES will minimise that effort by providing a generic kernel.

The rich set of connected tools and the Python interface will make possible to deliver a sort of grid based scientific data manipulation language.

GAGES can help in building the verification and validation benchmark data bases by providing tools for storing, transforming and analysing grid data. This will lower the costs of verification and validation of simulations, which is an important factor to make V&V a common practice.

Of course GAGES is not a miraculous solution and most likely due to the theoretical and practical constraints it will be only possible to cover restricted areas of application. For instance, in this thesis a whole universe of programs written in `FORTRAN` 77/90/2000, and tools to connect them with other languages, like `f2py`, is not considered. However one can imagine other GAGES-like environments, either based on different languages or different basic libraries, and the connection between these environments. In this way one can build a hierarchical structure of connected tools.

Chapter 6

Reuse of geometric models

This Chapter discusses the reuse of geometric models. As the simulations become more advanced, the importance of flexible and expressive geometry descriptions increases. For simple geometries it is feasible to resolve the geometry by hand while preparing data for mesh generators. However, with bigger and more complex geometries, handling them by hand quickly becomes impractical. Also for problems with unknown or changeable geometries one requires a suitable geometry description, as geometric properties must be updated while the simulation advances. All sorts of problems of parametric design and optimisation have also at their core the manipulation of geometries. Finally, the integration and automation of various stages of the engineering design process or scientific simulations is another push in researching the reuse of geometric models.

In the whole discussion about GAGES architecture the scope of the material presented in this Chapter is shown in Figure 6.1. After a brief discussion on the available geometric models, this Chapter proposes curvilinear boundary representation based on NURBS description as a practical foundation for implementing GAGES architecture related to geometric modelling. Further, this Chapter reports an investigation into the feasibility of the geometry bus of GAGES system, selecting the OpenNURBS package as an exemplar base library.

From the very board scope of geometric modelling problems this Chapter con-

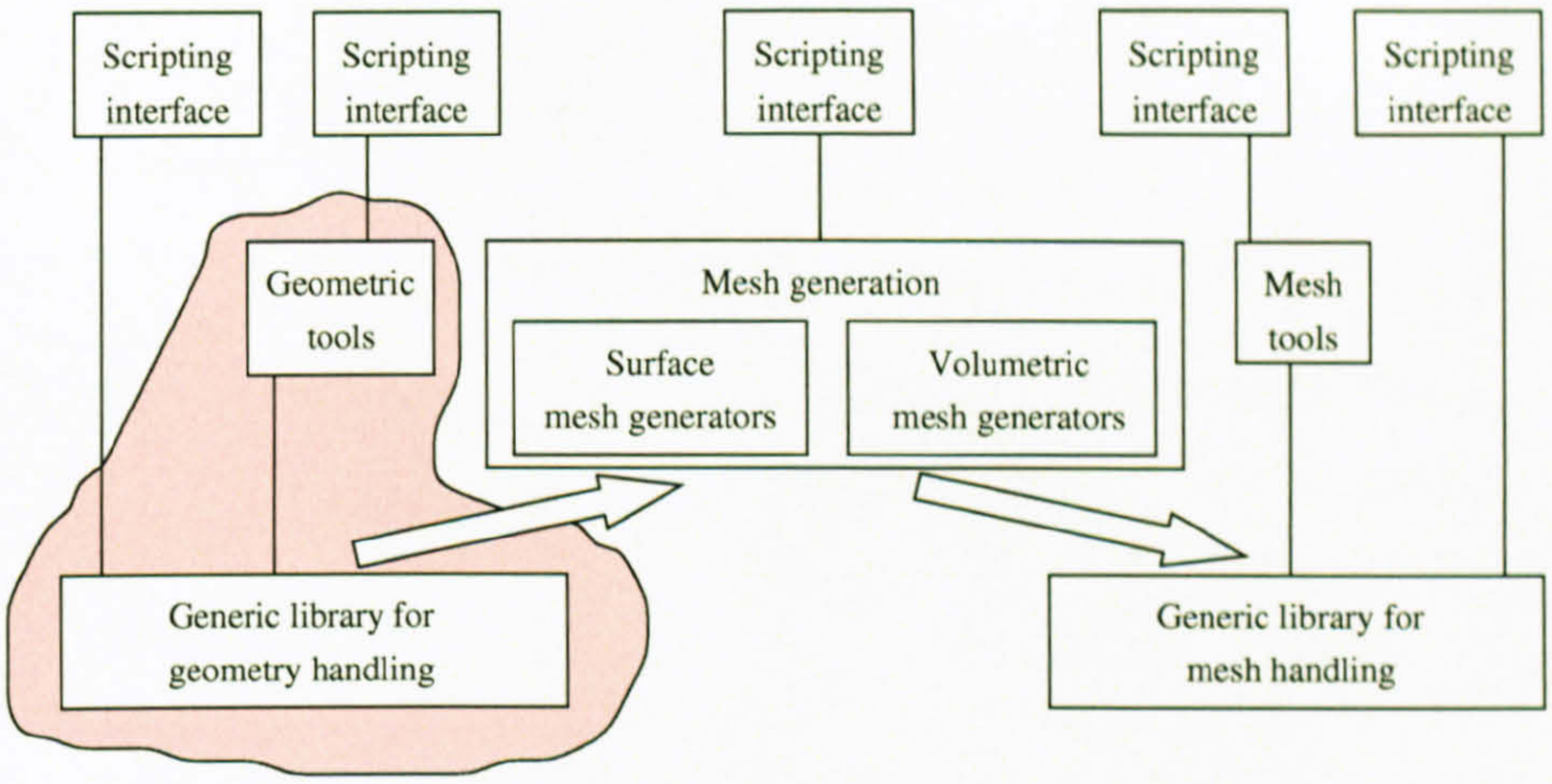


Figure 6.1: The scope of the material presented in this Chapter in respect to the whole GAGES architecture.

siders the issues related to the incorporation of geometric models into an in-house academic simulation system. The main motive will be the lack of suitable mechanism for linking large, both free and proprietary, software products based on geometric modelling with small scale special purpose research codes.

6.1 Basic representations of geometric models

There are many types of geometric model representations that are used in CAD, CAM and 3D graphics. Each of them exposes and stresses different features of the geometric model, and they were introduced with particular application areas in mind. In this thesis the focus is on geometric representations that lead to cellular decompositions of modelled objects, for instance finite element meshes.

With the above remark in mind, Figure 6.2 introduces a taxonomy of 3D model representations which is a modification of the one presented by Lin and Gottschalk in [2]. The modifications account for introducing the category of boundary representation models, and for treating the polygonal models as a specialisation of this category.

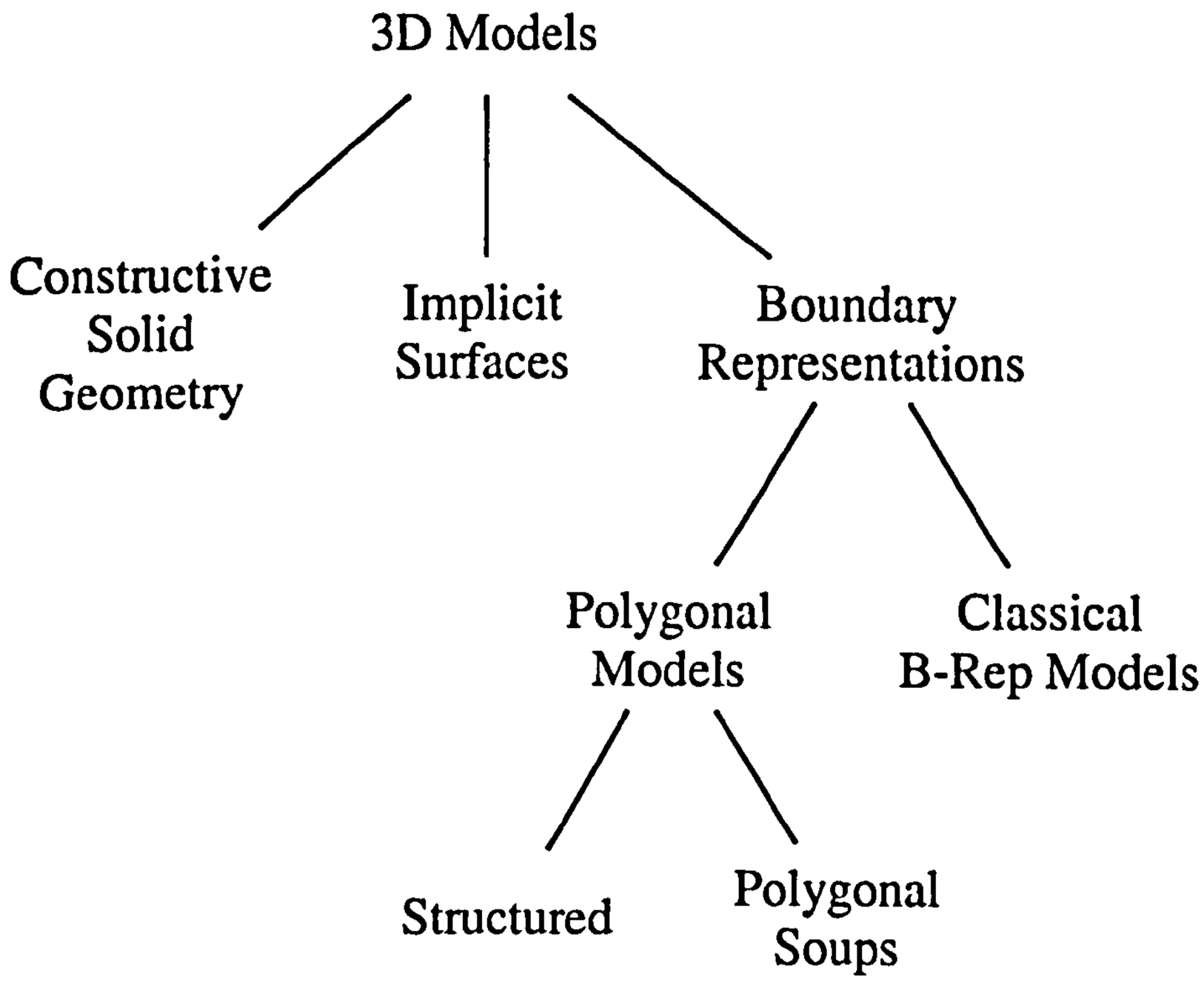


Figure 6.2: Taxonomy of 3D model representations based on the work of Lin and Gottschalk [2].

6.1.1 Implicit surfaces

Implicit surfaces are defined using functions of the form

$$f : R^3 \rightarrow R \ ,$$

which are maps from coordinates space to real numbers. An implicit surface is a set of points for which $f(x, y, z) = 0$. One desirable property of implicit surfaces is that they unambiguously define what is inside a model ($f(x, y, z) < 0$) and what is outside ($f(x, y, z) > 0$). Implicit surfaces are often used as primitives in CSG systems.

In the context of mesh generation implicit surfaces are rather seldom used, as it is quite hard to describe complex shapes using them. In the cases when they are applicable, one usually needs to transform them into a form of boundary representation. The exceptions are mesh generators based on octree decomposition where it

is possible to directly use implicit surface representation.

6.1.2 Constructive solid geometry

In *Constructive Solid Geometry* (CSG) solids are modelled as collections of primitive objects such as blocks, spheres, cylinders, cones and tori, combined by set–theoretic operations such as a union, intersection, and set difference [43]. CSG is very popular in CAD-CAM applications because even complex models can be easily specified by the users of a solid-modelling system. In the context of mesh generation CSG models cannot be directly used because the explicit representation of a model boundary is not available in a CSG model. Thus, as pointed in reference [113], CSG model must be converted into a boundary representation before it can be used in most of mesh generation systems.

6.1.3 Boundary representations

In a boundary representation geometry is given by explicitly specifying boundary elements such as vertices, edges, and faces, together with the topological relations between them. Boundary representations could be classified into two categories a) classical B-Rep representations where the topology structure is represented in terms of a kind of winged-edge data structure; faces can contain holes; general curves and surfaces are admissible, and b) polygonal models consisting of flat polygons where only a restricted set of topological relations is represented. The topology information very often consists of a face to vertices adjacency table only. In extreme, polygonal models could be just given as polygonal soups where only geometric information for each polygon is given, and there is no topological structure at all.

In the context of mesh generation boundary representations are the most convenient form because many mesh generation algorithms start with building polygonal approximations of object boundaries.

6.2 Geometric models exchange standards

In the previous section it was shown that there is no single geometric representation but there are several possible ways to describe geometry. Additionally, a single geometric representation can be encoded in files in many different ways.

Industry has developed several standards for exchange of geometric data. Some of the most popular ones are briefly presented below:

DXF: (Drawing eXchange Format) This is a proprietary format developed and maintained by AutoDesk and used in a family of AutoCad programs. This format is used for the exchange of drawings but it can also be used to read geometry data into special purpose analysis programs. The problem of DXF is that it is controlled exclusively by AutoDesk and the subsequent AutoCAD editions introduce often incompatible changes to the DXF format.

IGES: (Initial Graphics Exchange Specification) This is a format that provides definitions for the exchange of 2D and 3D product geometry, structure, relationships and annotations. It is used in drawing applications, finite element analysis programs and solid modelling software. IGES is supported by a very wide range of CAD/CAM programs. Its big advantage is that it supports migration to a more general STEP standard.

SET: (Standard d'Échange et de Transfer) This is a French Standard which includes finite elements, boundary representations, constructive solid geometry, scientific data and NC[1] tool paths. It was developed by Aerospatiale as a more compact alternative to IGES with similar features.

VDA-IS: (Verband der Automobilindustrie-IGES Subsets) This is a standard widely used both in the UK and German car industries.

[1]Numerically Controlled

JAMA: (Japanese Automobile Manufacturers Association) This association also defines its own subset of IGES standard.

STEP: (Standard for the Exchange of Product model data) STEP is an official ISO standard for exchange not only of geometric data, but all data involved in product design and manufacturing.

As can be seen, most of the standards have their roots in the IGES format. The current trend is however to slowly converge to STEP, which is believed to be the convergence point for CAD/CAM/CAE standardisation efforts.

6.2.1 Parasolid Pipeline and XT file format

Another example of common data exchange format is XT file format provided by the UGS company. This format is at the hart of the Parasolid Pipeline idea shown in Figure 6.2.1 and aims at a seamless integration of various CAD/CAM applications. Parasolid XT file format specification can be freely downloaded from the UGS web page `http://www.ugs.com/products/open/parasolid/pipeline.shtml`. While being a definitely valuable industrial solution this also is an example of a solution unsuitable for the academic environment. To take advantage of this format one has to implement a parser and other tools. This might be cost prohibitive, especially if it will be used for a single task. Generally, industrial solutions do not scale well down to academic environments, prevalent with single researchers or a small research group, prototyping, throw-away programming and special cases.

6.3 The need for a lightweight solution for the geometric model exchange problem

The software industry dealing with Computer Aided Engineering makes tremendous effort of integrating various tools for designing, manufacturing and management.

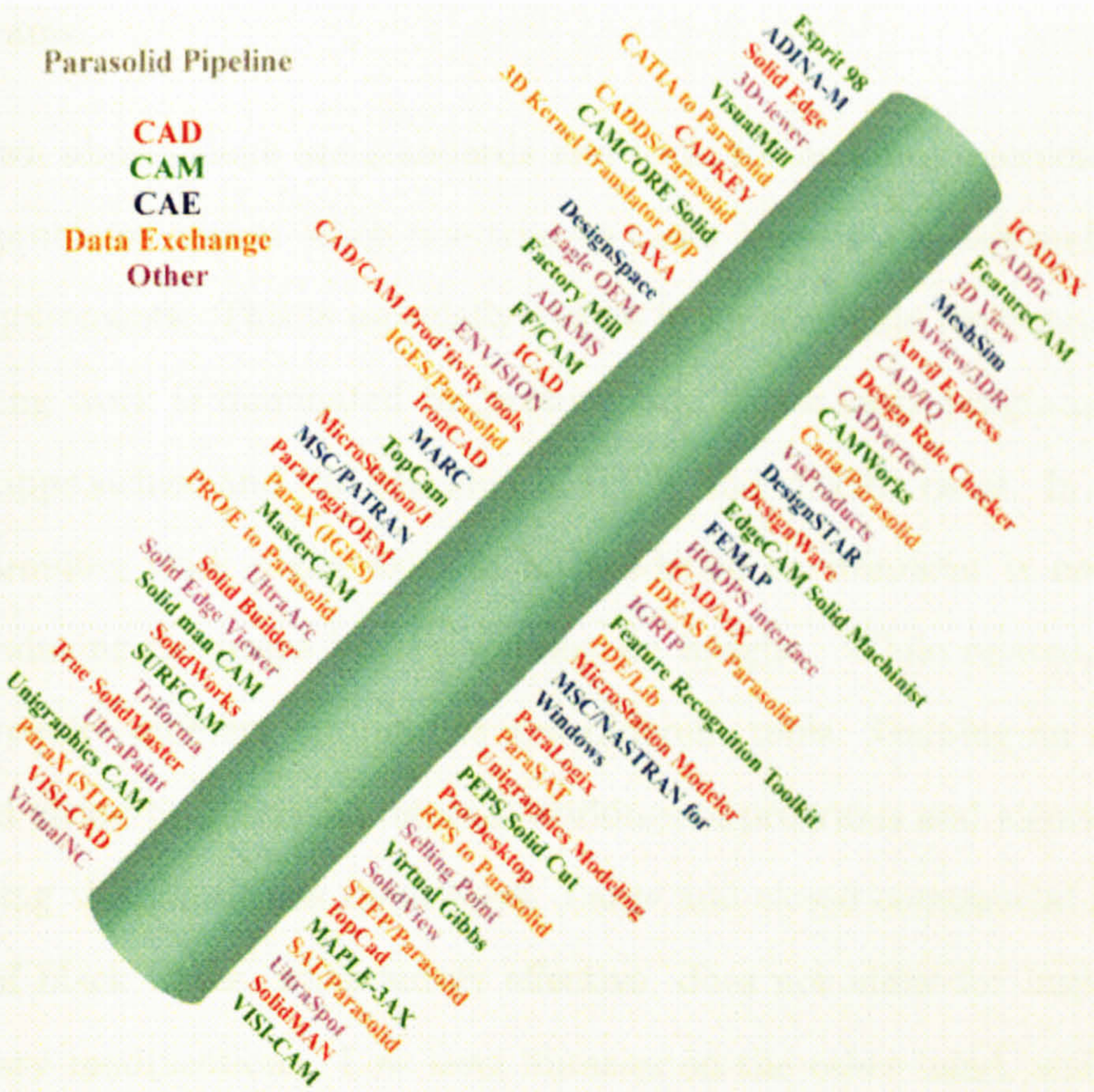


Figure 6.3: UGS Parasolid Pipeline [3]. Printed with permission

Thanks to this effort it is possible to exchange and reuse electronic data through the whole life cycle of a product, for instance starting from a conceptual design of the mechanical part, through its analysis using the finite element method, and finishing in the design of the machining process. Geometric models play in this data exchange an important role as they they are used as a reference point and a basis to which additional information is added.

Most of the time everything works smoothly, particularly if one stays within the family of commercial tools, especially from companies that cooperate with each other. However the situation changes when a new program from an individual is to be plugged into the pipe of processing tools. One may want to do this because such a tool chain offers a convenient interface which an in-house program lacks, or because one want to use an external tool as a reference point and compare the results of

both programs.

As shown above, there are standards and tools for handling geometric models. However, problems appear when one tries to tailor these standards and tools to a specific requirements. This is especially visible in an academic environment, where programming work is dominated by prototyping, throw-away programming, trial and error approaches, and the handling of single and specific cases. In most cases the programming work undertaken in an academic environment is not with the goal of producing programs but for gaining an insight. While related, these two activities require different approaches and different tools. Gaining an insight can be achieved either by observing internal workings of programs and algorithms or by implementing the algorithms themselves. Large and closed commercial software in the form of black boxes, while surely effective, does not allow for inspection and the necessary modifications. Low level libraries on the other hand, while allowing the incorporation of any observation mechanisms into the programs, require often building whole, complex applications, even if one is interested only with particular narrow problem.

Similar issues appear when one considers the use of commercial standards in academic environment. Firstly, these standards should be open, which is not always the case. Secondly, commercial standards are usually large and complex (vide STEP) so as to handle any possible case. Understanding them and limiting them down to suit the purposes of a particular project is usually very inefficient, and can yield incompatible solutions, as others solve the same problem in a different way. Finally, publishing a standard, even a good one, solves only a part of the problem, and even not the most important one. This is because the biggest problem and the bottleneck of effective exchange and manipulation of geometric models lies in implementation of various tools. This is the most time consuming and tedious activity, especially if one wants to do it in a portable and robust way, and to ensure that the tools will fit into the whole tool chain.

The above is the reason, that the author postulates that it is the most important to select appropriate geometric modelling API and its open implementation, before looking for data exchange standards. Firstly, bridging the APIs of different programs might be easier (one avoids the issues of parsing data files) and secondly, when it comes to implementing data file handlers, one is relieved from implementing the basic functionality over and over again. Of course the drawback of this approach is that there is no single API which would be applicable in all situations, or such an API would have to be either large and complex, or at very very low level, thus rendering it useless. However, if one is willing to sacrifice some code efficiency, good approximations to such an API can be found.

6.4 OpenNURBS as a basis for geometry exchange mechanisms

Finding a good geometry modelling library can be a problem too. Such libraries should be comprehensive but not to complex, complete, open source, with documentation, having some user community, desirably having some industrial support. Close to this ideal is library provided by the OpenNURBS Initiative [131]. On the OpenNURBS web page it is stated:

The OpenNURBS Initiative provides CAD, CAM, CAE, and computer graphics software developers the tools to accurately transfer 3-D geometry between applications.

The tools provided by OpenNURBS include:

- *A file format specification and documentation.*
- *C++ source code libraries to read and write the file format. Windows, Mac, and Linux are supported.*
- *Quality assurance and revision control.*
- *Various supporting libraries and utilities.*

- *Technical support.*

Because the OpenNURBS library supports manipulation of boundary representation models (B-Reps), and additionally has very extensive support for curvilinear geometry, it is a very good candidate to be a basis for implementation of the geometry bus proposed in Chapter 5. Support for the boundary representation models is important from the point of view of generation of volumetric meshes, for which the surface mesh is often the starting point. The handling of curves and surfaces via NURBS description is nowadays a standard in geometric modelling. However it should be stressed, that the choice of B-Reps and NURBS as the basis for geometric description is just one of the possible solutions. In fact, it is a trade-off between the genericity of description and the availability of implementation. The same way, OpenNURBS is just sample implementation based on these two foundations. From the general point of view of the methodology presented in this thesis, it should be possible to change OpenNURBS to another library, or even to base the geometric description on a different set of underlying concepts, for instance the one used in Djinn API [157].

Concrete programming tools provided by OpenNURBS library include:

- classes for handling 2D, 3D, 4D points, vectors and transformation matrices,
- classes for handling lines, planes, coordinate frames,
- classes for handling Bezier and NURBS curves,
- classes for handling Bezier and NURBS surfaces,
- classes for handling triangulation of surfaces,
- classes for handling B-Rep models,
- tools for handling colours, textures, lights, viewports,
- tools for OpenGL rendering,

- tools for persistent storage of geometric models,

- various mathematical tools.

However, there is one big problem (also a problem for other geometric libraries) which is the connection between geometric models and computational models based on grids. This connection is represented in GAGES as the connection between the geometry bus and the grid bus. The connection from the geometry bus to the grid bus is achieved via mesh generation. The connection in the opposite direction is achieved by keeping within the grid based model references to geometric entities. Although the OpenNURBS library provides data structures for handling surface meshes and classes representing B-Reps or their parts are aware of the meshes, OpenNURBS itself does not provide any mesh generation utilities, except for simple Cartesian meshes for tensor product NURBS surfaces.

In order to make support for geometry bus and its connection to grid bus possible and complete, it is necessary to implement the minimum the following functionality:

Adaptive discretization of 3D parametric curves: Sample implementation is presented in section 6.4.1.

Surface mesh generation: An implementation of a sample surface mesh generator based on `Triangle` Delaunay mesh generator is described in Chapter 9.

Discretization of B-Reps: This involves discretization of B-Rep edges, surfaces and volumes. One approach to the generation of volume meshes from B-Rep description is to generate suitable surface meshes taking into account geometric features and user specified mesh distributions, and then using this mesh to generate volume discretization. A detailed description of this process can be found in reference [22].

Beside the above tools it would be also necessary to provide some sort of geometry visualisation capabilities. The OpenNURBS library provides low-level support for

visualisation of geometric entities using OpenGL. However, it could be also desirable to allow direct visualisation of OpenNURBS geometric entities in high level visualisation environments such as either VTK or OpenDX. Section 6.4.3 describes a small library which allows the translation of OpenNURBS curves and surfaces to VTK data structures for subsequent visualisation.

According to GAGES concepts a library such as OpenNURBS is the basic fundamental layer necessary but not sufficient to provide time efficient solutions (not in the sense of CPU time but in total time from the concept to its implementation). In order to support interactive learning and effective tool building it is necessary to provide a scriptable (high level) language interface to the OpenNURBS library. Creation of such interface is described in Chapter 8.

6.4.1 Adaptive discretization of parametric curves

An important functionality missing from the OpenNURBS library is the discretization of parametric curves. OpenNURBS supports discretization only in uniform cases for which curve points are calculated in equal increments of the curve parameter. However, such discretization may be inappropriate, as it does not account for geometric properties of the curve. In areas of high curvature the point density should be high whereas more straight segments can be interpolated with fewer points. Such an adaptive discretization allows accurate resolution of geometric features, what is important in mesh generation, or when calculating geometric or mass properties such as length, inertia moments, etc.

Having in mind primarily a future application to B-Rep discretization, the OpenNURBS library was enriched during the work for this thesis with possibilities to adaptively sample parametric curves. Implementation of this feature is based on references [29, 51] and it permits the selection of one of four sampling methods:

a) **The sampling method based on segments length:** The concept of this method is depicted in Figure 6.4a). The segment ACB is considered to be an

accurate approximation of the curve if the following condition holds:

$$|d_1 + d_2 - d| \leq \varepsilon \ , \tag{6.1}$$

where d_1, d_2, and d are segment lengths and ε is a tolerance. In practice squared distances are used to avoid calculation of square roots.

b) **The sampling method based on angle between segments:** The concept of this method is shown in Figure 6.4b). The segment ACB is considered to be an accurate approximation of the curve if the following condition holds:

$$|\alpha - 180°| \leq \varepsilon \ , \tag{6.2}$$

where α is the measure of angle ACB and ε a tolerance.

c) **The sampling method based on chord distance:** The concept of this method is shown in Figure 6.4c). The segment ACB is considered to be an accurate approximation of the curve if the following condition holds:

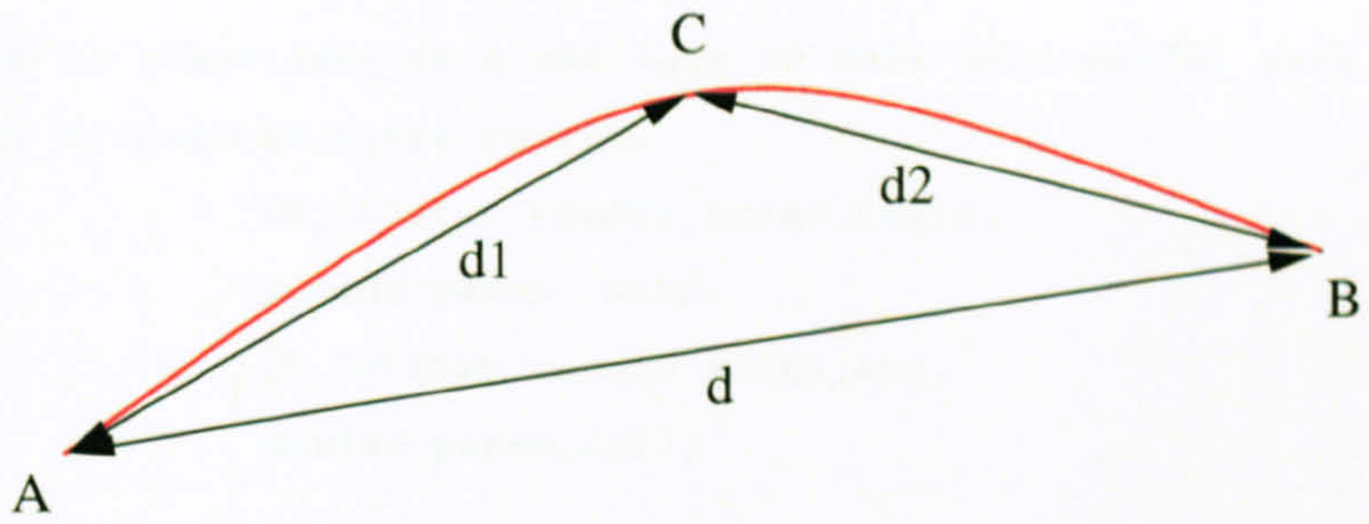
$$d \leq \varepsilon \ , \tag{6.3}$$

where d is the distance from the midpoint C to the chord AB.

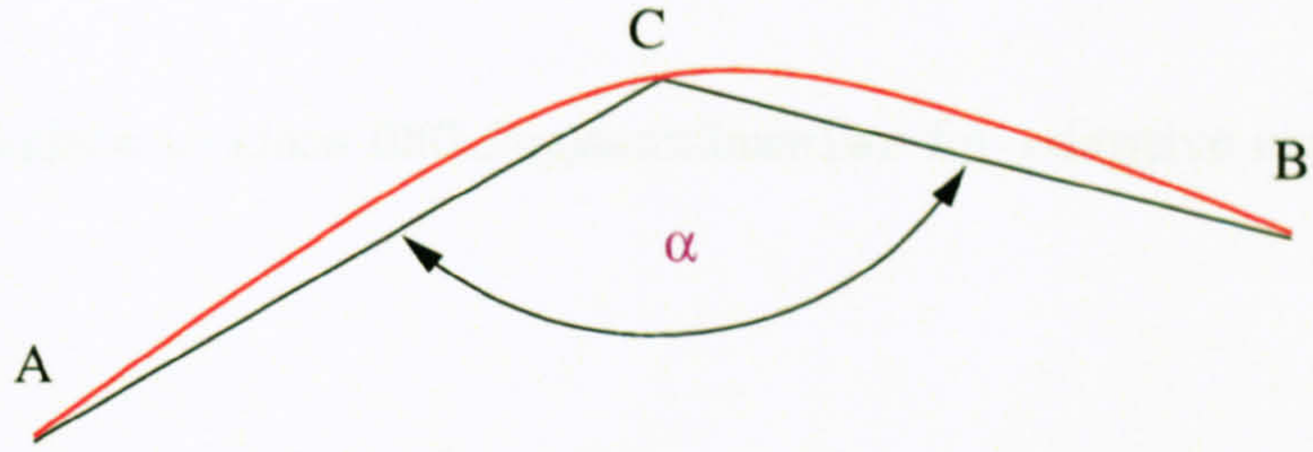
The midpoint C is calculated as the being in the middle of A and B in parametric space plus some random error introduced in order to avoid the aliasing phenomenon.

All the four sampling methods are implemented inside class `ONC_SegmentSampler` shown in listing 6.1.

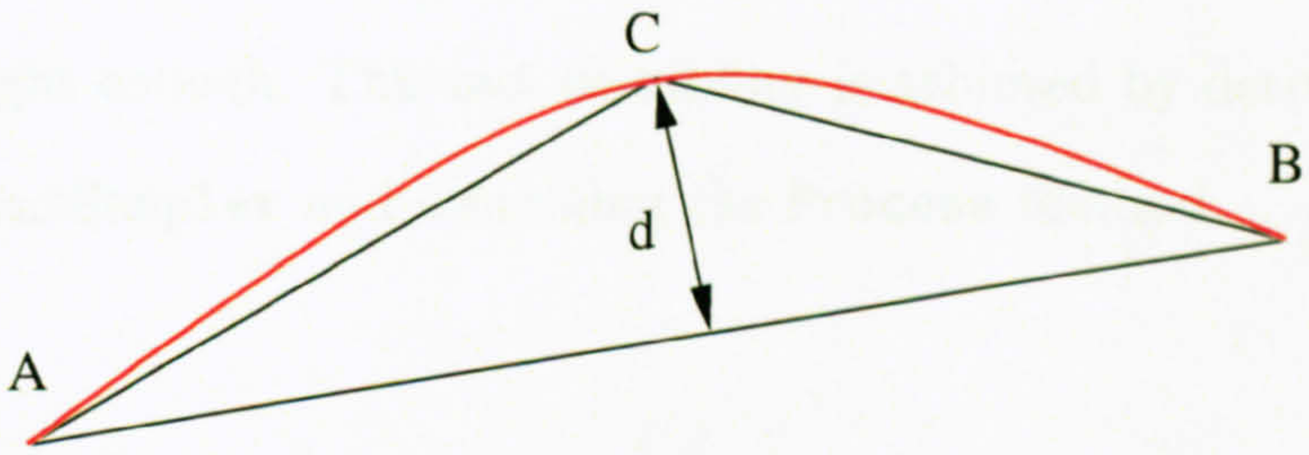
```
1  class ON_CLASS ONC_SegmentSampler {
2    public:
3      typedef enum {LENGTH, AREA, ANGLE, CHORD, UNIFORM} eTestMethod;
4
5      ONC_SegmentSampler() : tolerance(ON_DEFAULT_WORLD_TOLERANCE),
6                             counter(0) {SetTestMethod(CHORD);}
7      virtual ~ONC_SegmentSampler();
8
9      void SetSamplingTolerance(const double t);
10     double GetSamplingTolerance() const;
```



(a) Criterion based on segment length.



(b) Criterion based on angle between segments.



(c) Criterion based on chord distance.

Figure 6.4: Illustration of different criteria for adaptive sampling of parametric curves.

```
11
12      void SetTestMethod(eTestMethod m);
13
14      eTestMethod GetTestMethod() const;
15
16      int GetSamplingPoints(ON_3dPointArray &points, ON_Curve const &curve,
17                            ON_Interval *domain=NULL);
18
19      int GetSamplingParameters(ON_SimpleArray<double> &params, ON_Curve const &curve,
20                            ON_Interval *domain=NULL);
21
22      int SampleCurve(ON_Curve const &curve, ON_Interval *domain=NULL);
23
```

```
24      // This must be overridden in a subclass to make some useful work
25      virtual bool Process(ON_Curve const&,
26                           ON_3dPoint const& point_begin,
27                           double param_begin,
28                           ON_3dPoint const& point_end,
29                           double param_end);
30
31      unsigned int ProcessCount();
32  };
```

Listing 6.1: Interface to class `ONC_SegmentSampler` for adaptive sampling of parametric curves.

`ONC_SegmentSampler` permits either the array of sampled points or corresponding curve parameters to be obtained, or to perform any action on the segments considered to be straight enough. This last possibility is achieved by deriving a new class from `ONC_SegmentSampler` and overriding the `Process` method.

6.4.2 Calculation of surface principal stretches and principal stretch directions

Another enhancement to the OpenNURBS library, added during the work for this thesis in order to implement surface meshing, is the calculation of surface principal stretches and principal stretch directions.

The principal stretch directions are defined as the directions in parametric space (s, t) in which an infinitesimal linear element $\vec{dv} = (ds, dt)$ undergoes maximum and minimum elongation when mapped to the real surface.

The principle stretches and corresponding directions are sought by solving an eigenvalue problem for the matrix of surface first differential form, defined as:

$$\left(\begin{bmatrix} e & f \\ f & g \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \mathbf{v} = \mathbf{0} \tag{6.4}$$

with

$$e = \frac{d\vec{r}^T}{ds}\frac{d\vec{r}}{ds} \tag{6.5}$$

$$f = \frac{d\vec{r}^T}{ds}\frac{d\vec{r}}{dt} \tag{6.6}$$

$$g = \frac{d\vec{r}^T}{dt}\frac{d\vec{r}}{dt} \tag{6.7}$$

$$\vec{r}(s,t) = [x(s,t), y(s,t), z(s,t)] \tag{6.8}$$

where λ is stretch magnitude and $\mathbf{v}$ is associated stretch direction vector.

The above algorithm was implemented as function `ONC_EvPrincipalStretches`, with the following interface.

```
1 BOOL ONC_EvPrincipalStretches(
2         const ON_3dVector&, // Ds,
3         const ON_3dVector&, // Dt,
4         double *lambda1, // largest principal stretch value
5         double *lambda2, // smallest principal stretch value
6         ON_3dVector &L1, // lambda1 principal stretch direction
7         ON_3dVector &L2 // lambda2 principal stretch direction
8         );
```

Listing 6.2: Procedure for calculation of surface principal stretches and principal directions.

The calculation of the surface derivatives denoted above as `Ds` and `Dt` can be easily done using functions provided by the OpenNURBS library.

6.4.3 Linking the OpenNURBS library with the VTK visualisation toolkit

Other set of enhancements to the OpenNURBS library, that allow it to become closer to the idea of grid bus, are classes and functions for translating between OpenNURBS data structures and VTK [120] data structures. These classes and functions make possible to directly use OpenNURBS in VTK based visualisation programs. During the work on this thesis the following classes and functions were implemented:

`vtkONRevSurfaceSource` – This class is a filter which takes as an input reference to `ON_RevSurface` object and surface resolutions in axial and radial directions and produces on output `vtkUnstructuredGrid` discretization of a surface of revolution. Examples of the use of this class are given in Chapter 9.

`ON_CurveTovtkUnstructuredGrid` – This is a function to produce discrete view of parametric curve described via `ON_Curve` class. It takes two arguments, a reference to curve object and `ONC_SegmentSampler` object defining how the curve should be sampled.

`vtkONMeshSource` – This is a filter class that translates between `ON_Mesh` and `vtkUnstructuredGrid`. Examples of the use of this class can be found in Chapter 9.

6.5 Summary

This Chapter presented the discussion regarding the reuse of geometric models and the construction of the geometry bus for GAGES environment. Firstly, the overview of the basic categories of geometric models was presented. This was followed by the discussion on geometric models exchange standards. Based on this discussion this chapter postulated the need for a lightweight solution for geometric model exchange problem and indicated that this solution should be based on an appropriate geometric library. Among available libraries the OpenNURBS library was selected as the library which provides almost all the necessary tools for building the boundary representation models with curvilinear geometry. The features provided by this library were analysed from the point of view of linking the geometric bus and the grid bus together. This analysis showed that the main missing routines were routines related to discretization of the B-Rep models, for instance routines for discretization of parametric curves or routines for inspecting surface curvature properties. For the purpose of this thesis the missing routines were implemented and their implemen-

tation was presented in sections 6.4.1 and 6.4.2. Additionally few small utilities for linking the geometric tools based on the OpenNURBS library and the visualisation tools based on the VTK library were implemented and presented in section 6.4.3.

The next Chapter will provide a detailed discussion about the grid based components of GAGES. The ideas presented in this and in the next chapters will be further refined in Chapter 8 and discussed from the point of view of building a hybrid system.

Chapter 7

Reuse of computational grids

Chapters two, three and four explained the need for the integration of various components of scientific software, discussed possible ways of reaching that goal and advocated one technique in particular, namely software integration based on scripting languages, respectively. This Chapter concentrates on one particular aspect of software integration – *interoperability*. Reference [65] defines interoperability as the ability to understand and use data produced in one software component by another one. All data present in computer programs are organised into some structure, and in its essence interoperability boils down to translating between these data structures. The need for translating between data structures comes from:

- hardware differences – for instance big endian and little endian bits ordering,
- differences induced by programming languages – for instance row oriented versus column oriented arrays in C and FORTRAN,
- different ways of organising data in user applications.

In the case of low level data structures just involving streams of bits (communication protocols, basic data types in programming languages) more or less widely accepted standards are available, e.g TCP/IP, and tools like XDR library, helping to achieve interoperability. Without them the whole Internet could not be built. However,

in the case of the users' application data structures the situation is completely different. The difficulty of translating between the users' application data structures comes equally from the complexity and variability of these structures and from numerous incompatible implementations. Moreover, because users of these data structures form smaller and more fragmented communities, they do not have enough potential to successfully introduce standardised solutions. In fact, the main source of standardisation in the case of transferring complex data structures are commercial solutions, widely accepted either because of their technical superiority or more often thanks to a strong market position of their inventors. As an example one can point to the STEP standard or the DXF data exchange format (see section 6.2).

It should be noted here, that most of the standards and *de-facto* standards deal with off-core data transfer, that is they are build around the specification of exchange file format, e.g. Parasolid XT format [115].

This Chapter discusses interoperability issues in one particular case – exchange of grid based data structures. Grids are in author's opinion one of the two fundamental families of data structures used in scientific simulation codes (the other family of data structure are matrices).

The place of the presented material in the whole discussion on the GAGES architecture is schematically depicted in Figure 7.1. In order to support the feasibility study of the proposed methodology and the GAGES architecture, this Chapter discusses a sample implementation of the grid bus based on a selected gird handling library. It shows that this exemplar implementation meets all GAGES objectives in respect to handling grid data structures.

7.1 Ubiquity of grids

It would be appropriate before discussing the ubiquity of grids in scientific computing to give a precise definition of this term. However as pointed in reference [13] it is not possible to give one formal definition of the term *grid* that covers all its use.

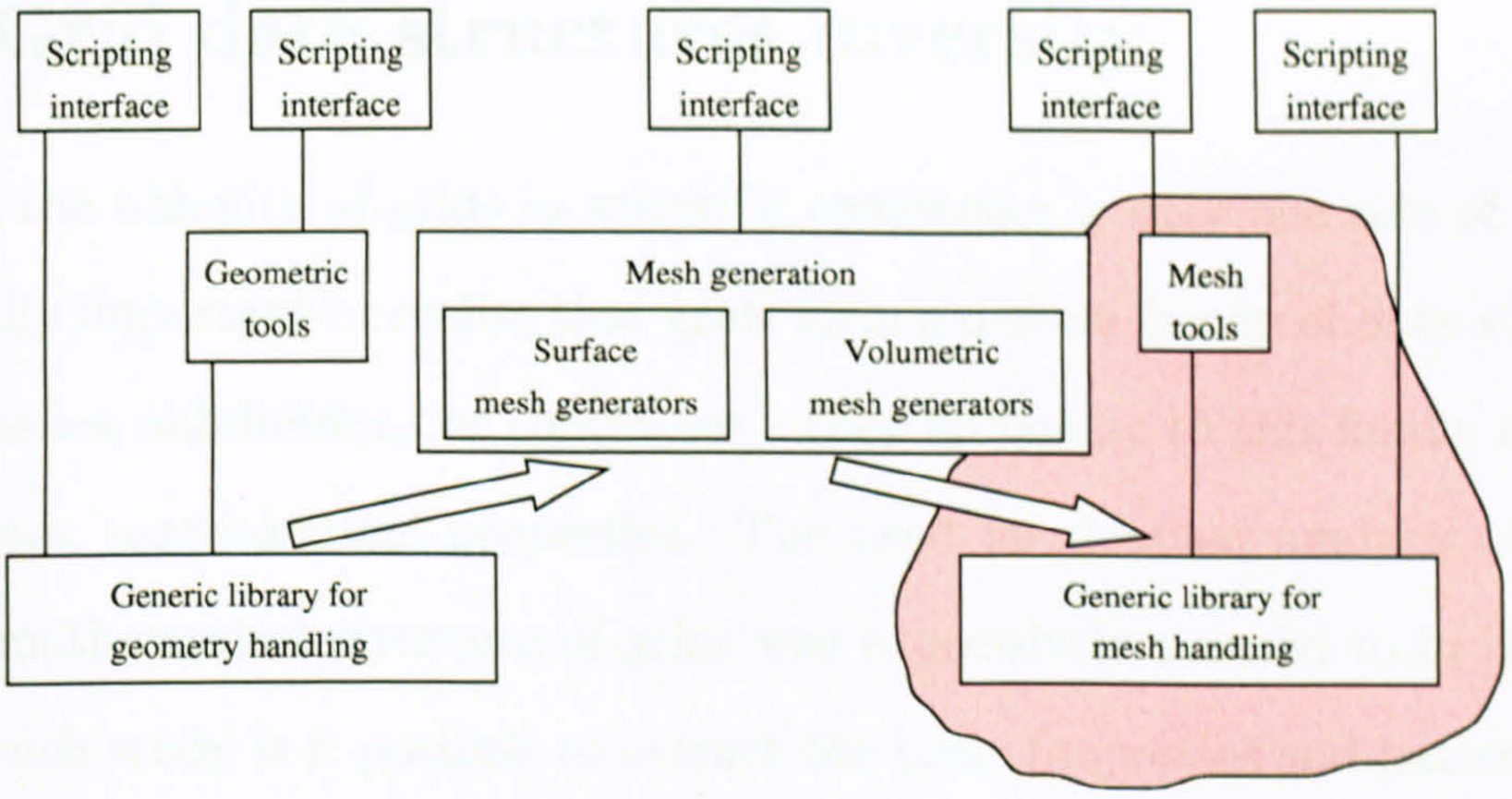


Figure 7.1: The scope of the material presented in this Chapter.

Moreover, in practice, grid abstraction is effectively used without fully realising rich mathematical structure behind it. The very vague description of a grid would be to say that it is a topological data structure over which scientific data sets are defined. Another description, which is closer to a common understanding of this term, is that grids are a way of describing complex geometric structures by locally arranging simpler objects, i.e. grid cells. In mathematics such arrangements are studied as cellular systems.

The description of complex domains as arrangements of simply shaped objects like triangles, quads, tetrahedra, or hexahedra, have proven to be really useful in constructing algorithms for finding numerical solutions of partial differential equations. All major numerical methods such as finite elements, finite volumes, finite differences or boundary elements use grids.

In the case of *Computer Graphics* grids allow the decomposition of complex objects into primitives which can be rendered on current graphics hardware.

Other fields that extensively use grids are *Computational Geometry* (grid generation), and *Computational Topology* and *Geometric Modelling* (all sorts of CAD/-CAM applications).

7.2 Grid data structures diversity

Realising the ubiquity of grids in scientific computing is only one side of the coin. It is equally important to realise that grids form a diverse family of data structures. Grids, meshes, subdivisions or complexes – they all belong to this family and share several basic mathematical properties. The need for detailed analysis of the underlying mathematical structure of grids was expressively pointed to in [13]. Only through such study is it possible to extract the basic properties and patterns which lead to generic implementations.

7.2.1 Variability of mathematical model

The first source of grid data structure diversity is the variability of their mathematical model. There are two main grid families:

- **structured grids:** for which topological relations can be implicitly expressed,
- **unstructured grids:** for which various topological relations must be explicitly stored.

In reference [13] a three-tier view of grids has been introduced. The grid functionality can be grouped into three distinct layers:

- **The combinatorial layer:** this layer only deals with the combinatorial properties of grids such as the adjacency relationships (element–nodes, element–edges, etc),
- **The geometric layer:** this deals with geometric realisation of grids, that is mapping from the space of the topological elements into a geometric space. An example of such a mapping is the assignment of point coordinates to grid vertices and the use of linear embedding,
- **The data association layer:** it deals with objects like grid functions or partial grid functions which are mappings from the space of the grid topological

vertex adjacencies	$M_i^0\{M^1\}$	$M_i^0\{M^2\}$	$M_i^0\{M^3\}$
edge adjacencies	$M_i^1\lfloor M^0\rfloor$	$M_i^1\{M^2\}$	$M_i^1\{M^3\}$
face adjacencies	$M_i^2[M^0]$	$M_i^2[M_\pm^1]$	$M_i^2\lfloor M^3\rfloor$
cell adjacencies	$M_i^3\{M^0\}$	$M_i^3\{M^1\}$	$M_i^3\{M_\pm^2\}$

Table 7.1: First order adjacency relations in grid.

elements into some space of user attributes.

The variability of combinatorial structure

In case of combinatorial properties, even when skipping fundamental difference between structured and unstructured grids, one can distinguish several grid data structures.

Reference [4] introduces a notion where M_i^d denotes i-th topological entity of dimension d. With some restrictions on the topology of a mesh (e.g. requiring faces and regions with no internal holes) it is possible to represent the mesh exclusively with the help of 0 to d-dimensional entities. For dimension d=3 these entities are:

$$T_M = \{\{M^0\},\{M^1\},\{M^2\},\{M^3\}\} \tag{7.1}$$

where M^d $(d = 0, 1, 2, 3)$ are the set of vertices, edges, faces and regions, respectively.

Having defined the topological entities one can define a set of first-order adjacency relations as the relations which describe for a given entity $M_k^{d_i}$ all of the entities M^{d_j} $(j \neq i)$ which are either on the closure of it $(j < i)$ or the ones which closure contains it $(j > i)$. If we introduce the following notation [4]:

$\{V^d\}$ is the unordered group of topological entities of dimension d,

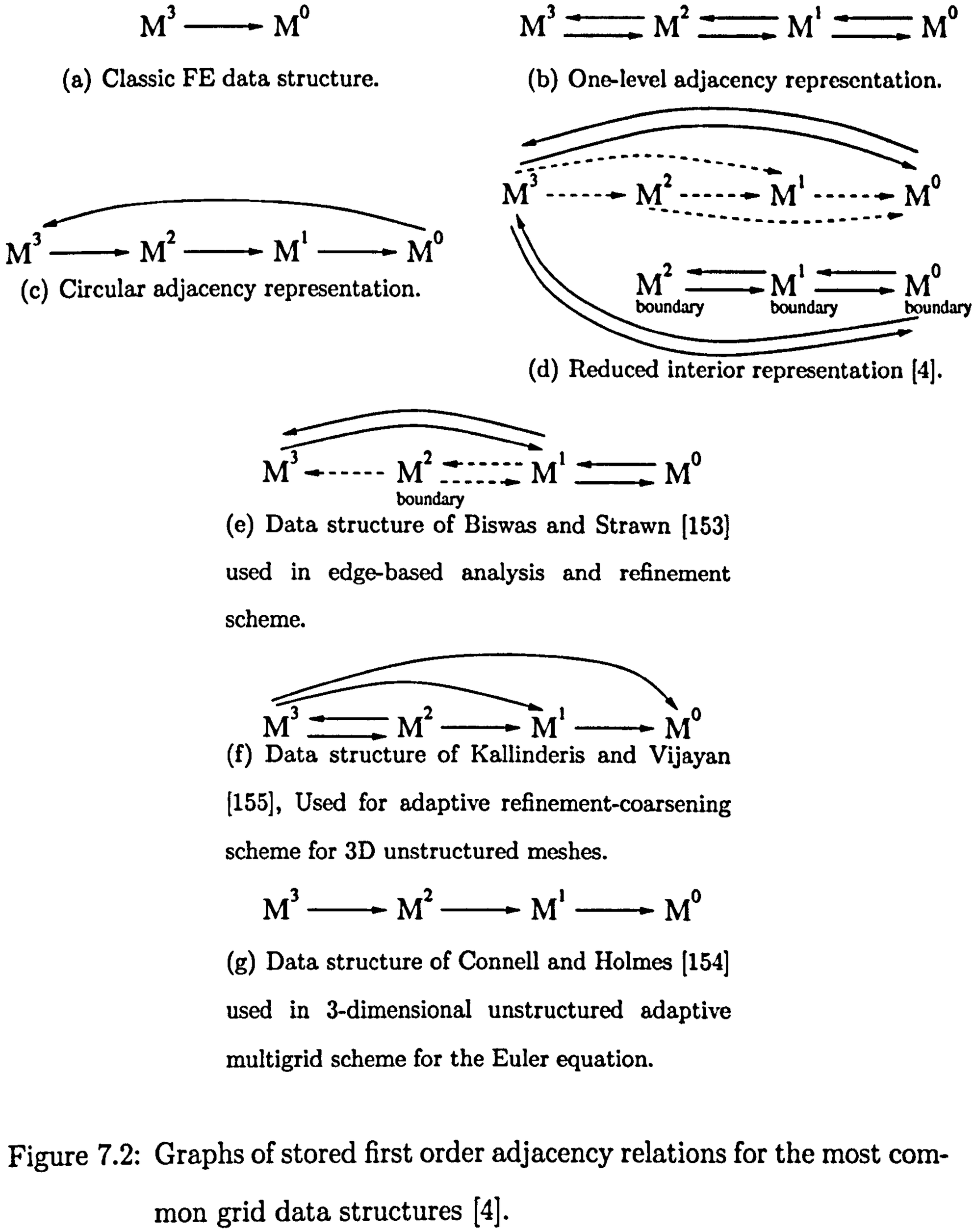
$\lfloor V^d\rfloor$ is the ordered group of topological entities of dimension d,

$[V^d]$ is the cyclically ordered group of topological entities of dimension d,

then the complete list of first order adjacency relations can be given as in Table 7.1.

Analogously one can define higher order adjacency relations.

Now, depending on the requirements, one can explicitly store different sets of adjacency relations. This will of course result in the differences in memory usage and the performance of a data structure. Different data structures can be conveniently depicted by graphs of stored adjacency relations. Such graphs for the most common cases discussed in reference [4] are shown in Figure 7.2.



(a) Classic FE data structure.

(b) One-level adjacency representation.

(c) Circular adjacency representation.

(d) Reduced interior representation [4].

(e) Data structure of Biswas and Strawn [153] used in edge-based analysis and refinement scheme.

(f) Data structure of Kallinderis and Vijayan [155], Used for adaptive refinement-coarsening scheme for 3D unstructured meshes.

(g) Data structure of Connell and Holmes [154] used in 3-dimensional unstructured adaptive multigrid scheme for the Euler equation.

Figure 7.2: Graphs of stored first order adjacency relations for the most common grid data structures [4].

The variability of geometric embedding

Also differences in the seemingly simpler geometric layer can lead to much different data structures. Structured grids can be embedded in the geometric space either by explicit coordinate specification or by implicit mapping. The implicit mapping can be specified for the whole domain or it can be derived from a parametric description of the domain boundaries (e.g. transfinite interpolation). Even geometries for unstructured grids, which seemingly allow only explicit specification, may be defined in an implicit way, with progressive meshes being the best example of this.

7.2.2 Diversity of implementations

From the above it should be clear that one can create numerous data structure models by combining various grid characteristics. It should be also obvious that each model can have multiple implementations with specific performance characteristics, that is memory usage, retrieval time, etc. In respect to the interoperability requirements, implementation details should not matter as long as the data structures provide the same functionality. In practice however, it is not so. This can be illustrated on two basic examples: the issue of node or element numbering and the issue of the order of vertices in cells.

While it is not strictly necessary, most of grid data structures are implemented with the assumption that grid vertices are mapped onto a set of positive integer numbers. This mapping is in the most cases hardwired into the data structure implementation (e.g indices of C arrays). The problem is that some implementations use 0-based indexing while other use 1-based indexing. This may sound as a trivial issue but it may cause "off one" errors, which as shown in reference [6] constitute a large share of the bugs in scientific codes.

The second issue – the order of the vertices in cells – can easily lead to bugs which either crash programs or, worse, produce nonsensical results. The order of nodes in cells is usually also hardwired into the implementation. The problem is that such

order is absolutely arbitrary. The very simple example is given in Figure 7.3 showing node orderings for a quad element. These orderings are topologically incompatible so care must be taken when copying grid data between structures which uses both of them. One can easily see, that with more nodes in the element, the number of

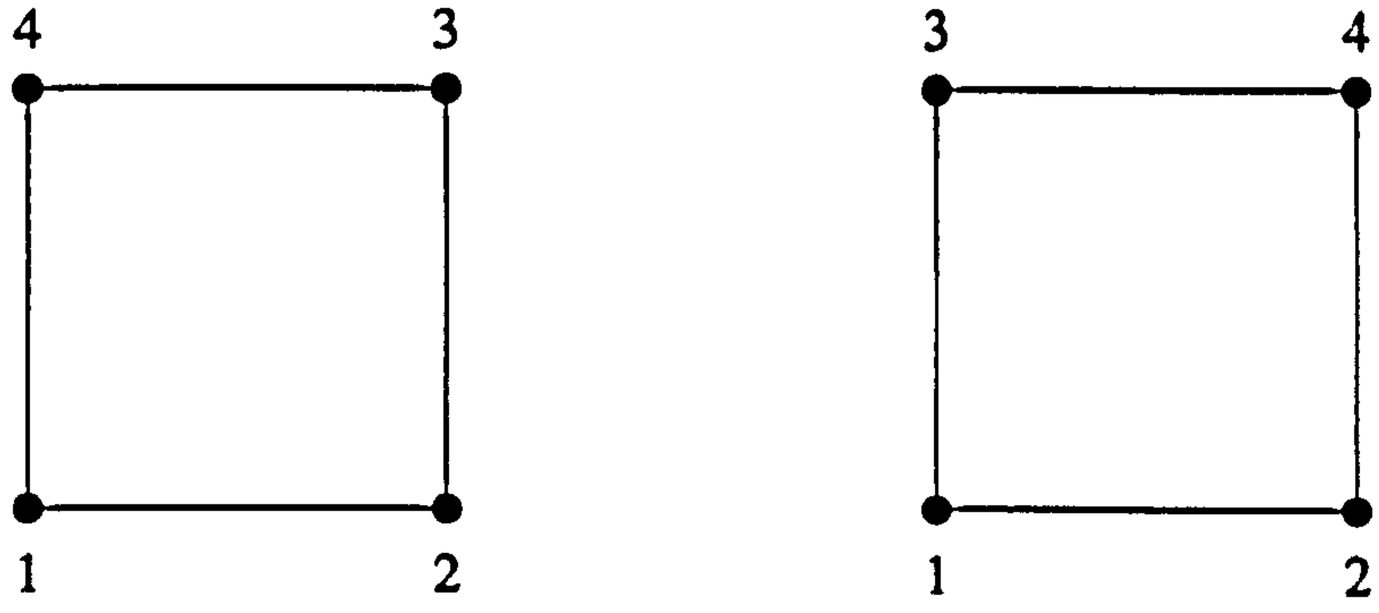


Figure 7.3: Possible ordering of nodes in a quad element.

possible orderings increases. In many simulation codes and grid handling libraries ordering of nodes in cells is a fixed property of the data structure and should be carefully checked and strictly obeyed. The only library known to the author which can transparently handle different orderings is `GrAL`.

7.3 Exchange of grid based data

The two previous sections showed two facts: the ubiquity of grids in scientific computing and the diversity of grid data structures. Having in mind the discussion from Chapter 2, concerning the need for software integration, it should be clear that the enabling exchange of grid based data is an important as well as a non-trivial problem.

7.3.1 Requirements for grid data exchange mechanisms

Designers of grid data exchange mechanisms have to take into account several, often contradictory requirements:

- openness,
- extensibility,
- portability,
- efficiency,
- maintenance and data safety.

In further discussion in this thesis the term "Grid Data Exchange" will be denoted by the GDE acronym.

Openness

This is a non-technical requirement as it concerns dissemination of grid exchange technology. In a world where most of the software components will be developed by third parties, grid data exchange must be based on unrestricted mechanisms. This includes open specifications, sample implementations with access to source code, and appropriate documentation.

Beside realistic technical design, producing software must be based on a realistic economic model, in order for the software to be ever written. From the perspective of the dissemination of grid data exchange mechanisms the most suitable would be distributing the technology as Open Source. Providing a technology as an Open Source allows unrestricted adoption of it in the users' projects, without the fear that the projects will be influenced or controlled by the technology provider. Contrary to the strict commercial solutions, including for example patenting of algorithms, sharing the technology as the Open Source is in the spirit of unrestricted sharing of scientific results. The last but not least is the financial issue – Open Source allows individuals or research groups the access and experimentation with a technology without the licence fee constraints. An important factor in the above is building a devoted users community which can sustain continuous development of the technology.

Extensibility

It is not possible to propose a single data exchange mechanism which would suit all users. However, it is important to make any such mechanism as extensible as possible. Extensibility is here understood as the ability to apply the GDE mechanism to new application areas, as well as the possibility to derive new GDE schemes tailored to specific user requirements. In practice, it means reliance on generic solutions, postponing concrete choices as long as possible, and shifting most of them to the user space.

Portability

In the situation when computer networks link very heterogeneous computing environments, portability is an important requirement, though the one which is most difficult to fulfil. Generally, portability requires solutions which account for differences in:

- the type of operating system,
- the versions of the operating system,
- the programming language, and
- the programming environment (e.g. compiler vendor, compiler version).

In case of GDE mechanisms, both data centric solutions and application centric solutions will have to deal with portability issues. Porting GDE mechanisms based on the specification of data formats and protocols should be a little easier, however regardless of which solution is adopted, taking portability into account increases development costs considerably.

Efficiency

If grid data is translated between different formats only once per simulation run, then translation efficiency is often not an issue. However, in a very heterogeneous and diverse simulation environments grid data might frequently have to be

exchanged between components based on different grid formats. In such cases, inefficient data exchange mechanisms may turn out to be the bottleneck of the whole system.

A carefully designed GDE mechanism should take into consideration that often only a part of the grid data needs to be transferred, e.g. only vertex geometry. The users of GDE should have board control over what aspects of grid data are exchanged.

Maintenance and data safety

Taking into account that a particular GDE could be used for a prolonged time, one should also carefully consider the issue of GDE maintenance. This requirement concerns both the design side (e.g. by choice of a self describing data format) as well as the implementation side (for instance proper documentation).

In some systems, for instance meteorological or astronomical, grid based data can be stored for several decades. Data safety in such cases means that data can be retrieved and interpreted even if there were numerous changes in computer technology in the meantime. On a small scale it also means, that research group's data does not become useless after the person knowing the quirks of the data format or API leaves the team.

7.3.2 Off-core versus in-core exchange of grid data

In this thesis two general categories of grid data exchange mechanisms are introduced:

- off-core exchange, which relies on external storage, and
- in-core exchange, where data structures are built in RAM.

Off-core exchange of grid based data: In the case of the off-core exchange of grid based data the data is saved in a special format in an external storage device.

Two parties wanting to exchange data in this way have to parse and interpret data file(s). This way of exchanging data seems to be most universal, however there are a few problems with it. Firstly, there is no widely accepted standard for saving grid data. Basically, each major software package uses its own format. There are some attempts to introduce more uniform solutions based on HDF5[1] or XML[2] but they are not widespread or lack appropriate tools. Another problem with off-core exchange is that it is difficult to provide partial views of grid data – often the whole file has to be parsed and only then can an appropriate view be set up.

In-core exchange of grid based data: In the case of in-core exchange of grid data the data is exchanged between data structures residing in the computer's memory. Basically, this kind of exchange consists of bridging APIs of respective data structures. This way of exchanging data is very flexible, it naturally allows partial views or computation of the necessary information on the fly. It is also faster than off-core exchange as there is no file parsing and external storage access. The biggest disadvantage are tighter bounds between applications.

Both ways should not be seen as opposite but complementary solutions. Off-core data exchange is mostly used when transferring data between remote machines, though it is possible to simulate in-core transfer with the help of RPC, CORBA or similar[3]. It is also used when persistent storage is required, but that can be achieved using an in-core scheme via persistent data structures. In-core exchange is mostly used in case of multi-threaded or multi-component applications.

It was said above that off-core and in-core exchange are complementary solutions. However, in the light of component interoperability this thesis claims that in-core data transfer plays more practical role.

Provision of, even a very versatile, data format solves the problem only partially.

---

[1]Hierarchical Data Format, version 5.

[2]eXtensible Markup Language.

[3]Because data has to be transferred via a network it has to be saved in some off-core data format.

Leaving users with data format alone, forces them to implement file parsers themselves. This is often beyond users' time constraints or capabilities. On the other hand, when some sort of standard parser is provided then the case of off-core transfer boils down to in-core transfer as it will be translation between parser API and user's data structure API.

The rest of this Chapter is devoted to the case of in-core data transfer remarking on off-core transfer when necessary.

7.4 Exchange of grid based data in context of multi-language programming

Multi-language applications do not alter significantly the picture of grid data exchange. In the case of off-core data transfer introduction of a scripting language may help in building grid data file parsers, as most of the scripting languages are endowed with extensive support for text and files processing. An example of this could be a multi-language application build by extending AWK with C, which takes advantage of AWK text file parsing capabilities, and C for providing complex data structures.

The case of in-core data exchange still reduces to translation between APIs, but additionally involves an intermediate layer of scripting language API. Additionally, there is an advantage in providing a scripting language API to data structures, because scripting languages have often very convenient interfaces to high level data structures like lists, dictionaries, or trees, which can help in interactive manipulation of grid data.

7.5 Universal grid exchange layer

7.5.1 Grid handling libraries

In this point some grid handling libraries will be discussed. It is practically impossible to discuss all available libraries, and it is out of scope of this dissertation to discuss most of them. Instead, four specific implementations will be described which can be treated as representatives of different classes of grid handling libraries. The libraries will be presented in the order from the simpler to the more complex in terms of abstractions, design and programming constructs.

E-Lib

E-Lib is a simple mesh handling library accompanying the book [5]. It was developed at Structural Engineering Computational Technology research group at Heriot-Watt University to avoid duplication of effort when dealing with common mesh operations. The library was developed having in mind finite element applications. The library is build around a classical finite element mesh data structure, that is, it stores coordinates of nodes and a element–to–node adjacency table. The main data structure of E-Lib is MESH and Listing 7.1 shows selected elements of it:

```
  typedef struct
{
   char *Title;                          /* title */
   int NMeshPoints;                      /* number of mesh-points */
   int NNodes;                           /* number of finite element nodes */
   int NElemsLink1;                      /* number of LINK1 elements */
   int NElemsTriang1;                    /* number of TRIANG1 elements */
/* Other elements skipped. In total there is 13 types of elements */

   int TotalNElems;                      /* total number of elements */
   int NBCNodes;                         /* number of boundary condition nodes */
   int NLoadedNodes;                     /* number of loaded nodes */
   int NMats;                            /* number of materials */
   int NMdfMats;                         /* number of .mdf declared materials */
   int NCompMatsType1;                   /* number of type 1 composite materials */
   int NTimeSteps;                       /* number of time-steps */
```

```
int NInternalTimeSteps;                 /* number of internal time-steps */
int NExternalTimeSteps;                 /* number of external time-steps */
int FirstNetIndex[E_NELEM_TYPES];       /* first net index of each element type */
double TimeStep;                        /* time-step */
double DampingFactor;                   /* viscous damping factor */
double Beta;                            /* Newmark's beta integration constant */
double Gamma;                           /* Newmark's gamma integration constant */

int NStressPointsLink1;                 /* number of stress points in LINK1 */
int NStressPointsTriang1;               /* number of stress points in TRIANG1 */

double **MeshPointCoords;               /* mesh-point coordinates */
int **NodesLink1;                       /* LINK1 node indices */
int **NodesTriang1;                     /* TRIANG1 node indices */
int TotalNodalDOF;                      /* total nodal degrees of freedom */
int *NodalDOF;                          /* nodal degrees of freedom */

int  *BCNodes;                          /* boundary condition nodes */
int **BCTypes;                          /* boundary condition types */
double **BCDispls;                      /* initial nodal displacements */
double **BCSpringConsts;                /* spring constants */

int *LoadedNodes;                       /* loaded nodes */
double **Loads;                         /* loads */

void **Mats;                            /* materials */
MAT  **MatsLink1;                       /* LINK1 materials */
MAT  **MatsTriang1;                     /* TRIANG1 materials */

int *NodeTypes;                         /* remeshing node types */
int *NNurbsCurves;                      /* number of NURBS curves per node */
double **NurbsCurveParams;              /* NURBS curve parameters */
NURBSCURVE ***NurbsCurves;              /* node NURBS curves */

int *GNIs;                              /* global node indices */
int *GEIs;                              /* global element indices */
int *GEIsLink1;                         /* LINK1 global element indices */
int *GEIsTriang1;                       /* TRIANG1 global element indices */

GEOM Geom;                              /* geometric model */
MAT *Props;                             /* material properties */
COMPMAT1 *CompPropsType1;               /* type 1 composite material properties */
DECOMP Decomp;                          /* decomposition data */
```

```
    MESHPARAM MeshParam;                    /* mesh parameters */
    FEERROR FEError;                        /* finite element errors */
    STRESSES Stresses;                      /* finite element stresses */

    int KeywordSet[E_NKEYWORDS];            /* keyword Booleans */
    int SubStructSet[E_NSUBSTRUCTS];        /* sub-structure Booleans */
}
    MESH;
```

Listing 7.1: Partial definition of E-Lib MESH data structure.

E-Lib has been successfully used in many projects including mesh generation, load balancing and mesh partitioning, membrane structures design, etc. Its main advantage is simplicity, which allows beginners to quickly pick up the basic concepts and to use the library in their applications. However, E-Lib shows also some drawbacks. Probably the most severe one is that, while allowing meshes with different element types, it does not provide an uniform API to operate on elements. Element data is divided into separate tables and that forces users to write separate loops for each table. Other restrictions are: the static nature of the MESH data structure, a fixed set of element types, no support for attaching data to nodes or elements. These restrictions are however the result of the simple nature of this library and it should be said that in its class the library is a very useful tool. E-Lib is implemented in a clean way in ANSI-C and is fairly portable.

GTS

GTS stands for GNU Triangulated Surfaces. As the name indicates it is a library for handling triangulated surfaces. The library provides dynamic, highly flexible data structures with interface to access and iterate over nodes, edges, faces. The library provides an API to handle geometric and topological modification of surface triangulation, e.g. progressive meshes with refinement and coarsening possibility, a 2D incremental Delaunay triangulation algorithm, or Boolean operations on triangulated surfaces. The main restriction of this library is that it only handles linear

triangular elements. GTS is implemented in ANSI-C but it uses the GLib object system to express object oriented design in C.

E2-Lib

E2-Lib was designed as a successor to E-Lib, with intention to fix E-Lib deficiencies and to provide a more universal tool. The primordial impulse for writing E2-Lib was the observation that at the core of many grid operations lies the manipulation of small sets of integer numbers. Those integer sets express either adjacency relationships between grid elements or are used in the tagging system which links grid elements with physical properties such as geometric objects, materials, boundary conditions, etc. Though not rigorously proven, it was demonstrated on several examples that calculation of adjacency relations as shown in table 7.1 can be very succinctly expressed in terms of basic set-theoretical operations like union, intersection, difference, symmetric difference, closure. For this reason one of the first data structures implemented for E2-Lib was data structure called `e2_Iset` (from "integer set") which is basically an "intelligent" integer vector endowed with some knowledge about its state – together with integer elements this structure holds information about the size and integer flag indicating if the vector is sorted, how sorted (ascending, descending), continuous (i.e. contains consecutive integers) or what is the mode of iterating over its elements (linear of cyclic). The scope of this additional information was selected in such way that it can be squeezed into an integer number, so internally Isets can be treated as continuous integer arrays. Keeping and updating this additional data introduces some overhead, but it is far outweighed by the gains from being able to optimise operations on Isets.

The next logical step in implementing E2-Lib was the introduction of data structure for holding collection of Isets. This structure was called `e2_CollIset` and was implemented as dynamic array of Iset pointers. With these two data structures and supporting functions, it is possible to represent all grid adjacency relations and

effectively implement topology manipulation routines.

In order to represent finite element meshes it was necessary to provide a way to store mappings from topological elements into spaces of real numbers (for instance to store nodes coordinates). In E2-Lib this role is played by `e2_Field` which effectively permits the storage of tensor fields of any rank and dimension. The `e2_Field` was implemented as a 2-dimensional dynamic array with one dimension fixed at the time of creation. This structure is compatible with `e2_Iset` in the sense that Isets can be used to describe samples of Fields.

Other data structures implemented for E2-Lib include a unified representation of boundary conditions and loads, able to represent Dirichlet, Neumann and Robin boundary condition types, a dynamic table based on red black threaded trees, dictionary like data structure for mapping from strings into real or integral values, and open flexible data structures for holding entire finite element models [4].

E2-Lib is a solid piece of work and has several advantages, however it also has one great disadvantage – it is not wrapping friendly. At the very core of E2-Lib lies the `e2_Iset` data structure and due to a decision which now could be described as a design flaw: it is very volatile. Isets are manipulated through pointers to `e2_Iset`. However, such pointers cannot be permanently stored, because due to dynamic nature of Isets, subsequent operations on them may invalidate them. The basic idiom for working with Isets is show below:

```
e2_Iset *foo;
...   // initialise iset
foo = e2_IsetAdd(foo, 23) // adds 23 to the set of integers
```

The selected way of operating on Isets allows the creation of very efficient C code, but practically precludes wrapping E2-Lib in a scripting language.

[4] E2-Lib was designed and implemented by Dr Peter Iványi and the author. Although the author has in some sense abandoned his creation lured by the GrAL library, E2-Lib is further developed and actively used by the second of its designers.

GrAL

The GrAL (Grid Algorithms Library) is an open source C++ library implemented by G. Berti [13, 135]. The main goal of GrAL is genericity and reuse of grid manipulation algorithms: "Its aim is to provide grid-based algorithms in a way that is completely independent of the data structures used to represent grids" [13]. GrAL has solid theoretical foundations described in reference [13] and is based on a carefully selected set of abstractions (much like the Container and Iterator concepts for STL) allowing grids to be described regardless of their concrete implementation. The hierarchy of these concepts is shown in Figure 7.4. Another fundamental GrAL

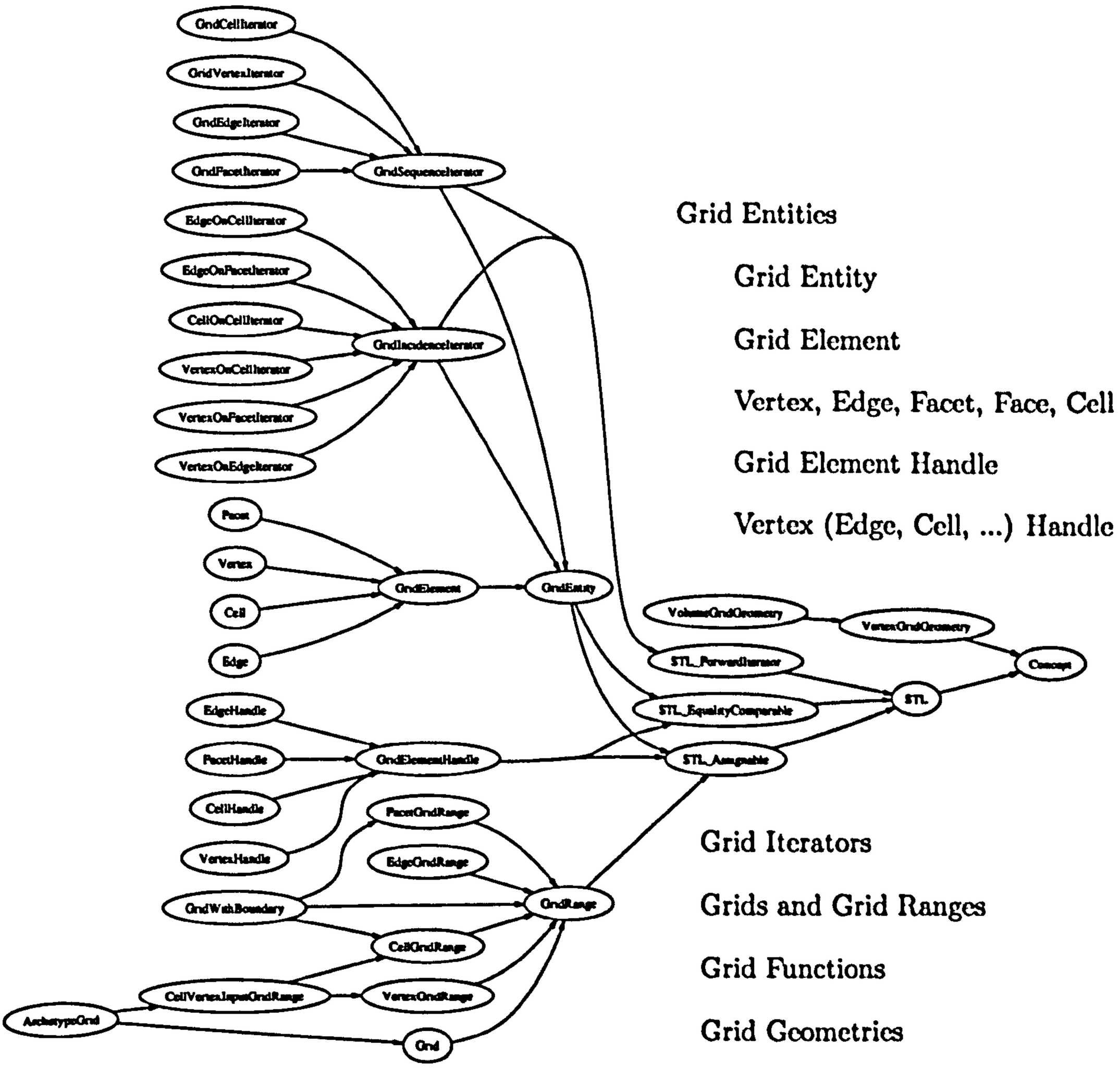


Figure 7.4: GrAL concepts hierarchy

property is the separation of the grid's functionality into three distinct layers:

1. **Combinatorial layer:** In this layer grid is seen from purely combinatorial point of view as a lattice given by incidence relations.

2. **Data association layer:** As a consequence of this layer it is possible to associate data with discrete grid elements. A simple example of such association is numbering of elements, a more complex is a velocity field prescribed in each vertex.

3. **Geometrical layer:** This layer provides embedding into concrete geometrical space. It deals with things like vertex coordinates, distances, volumes, etc.

7.5.2 GrAL adapters

GrAL offers a set of concrete implementation of grid data structures:

`Triang2D`, `Triang3D` – simplicial grids,

`Complex2D`, `Complex3D` – arbitrary polytope grids,

`Cartesian2D`, `Cartesian3D` – structured grids,

`ComplexND`, `CartesianND` – grids of arbitrary dimension, also run time specified.

These data structures offer trade-offs between processing speed, memory usage and the set of supported algorithms.

The real power of GrAL lies however not in the number of provided implementations, but in the fact that the library is highly generic. Thanks to template programming, algorithms do not depend on concrete data structures. Thus, it is possible to use any concrete data structure with the GrAL algorithm, provided that one creates an interface layer satisfying the syntax and semantics defined by the GrAL concepts. That interface layer will be called the GrAL *adapter*.

Figure 7.5 shows the situation when GrAL is used to connect a custom grid data structure with a grid manipulation library, here a partitioning library. In this case

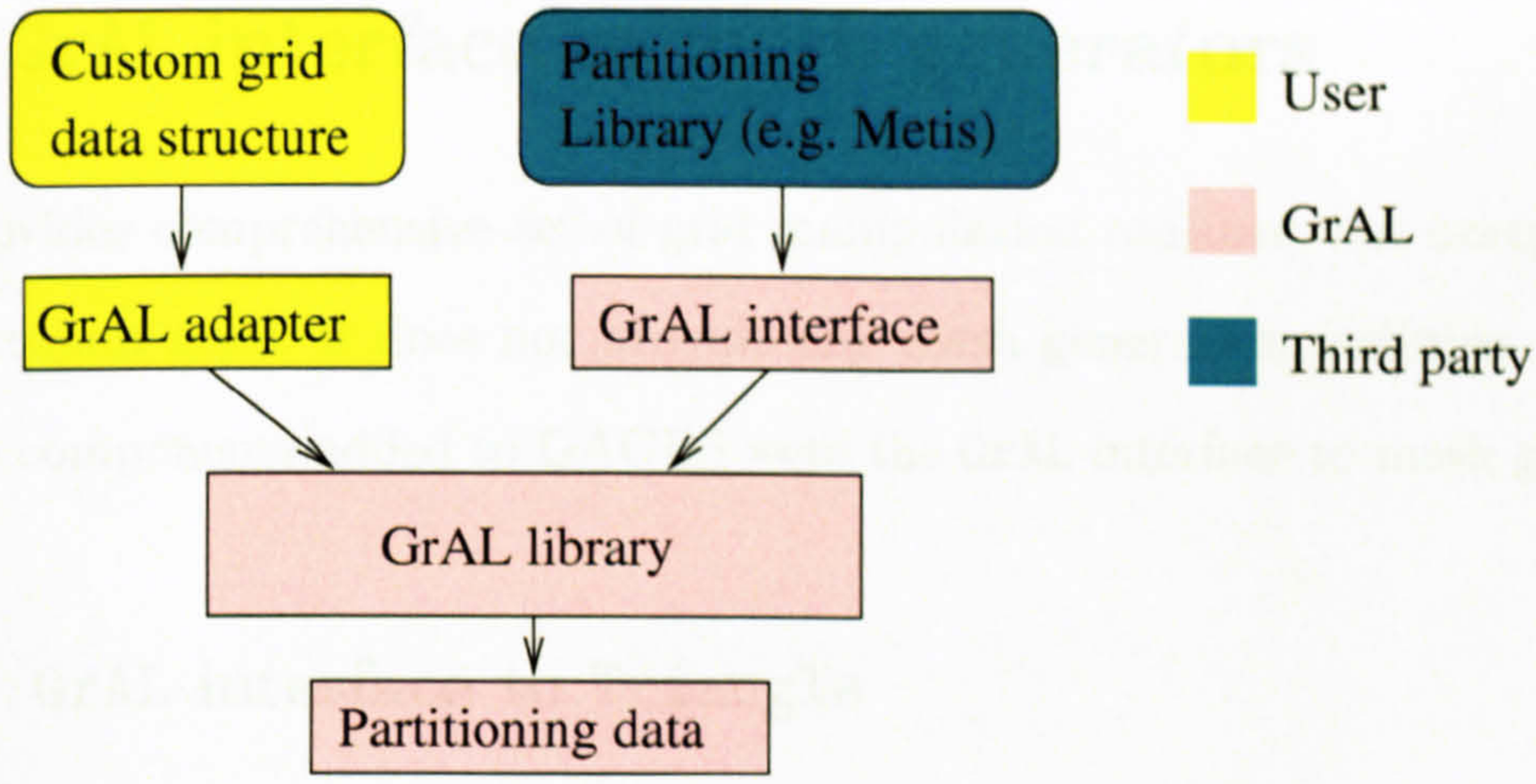


Figure 7.5: GrAL used to interface a custom data structure with a third party library.

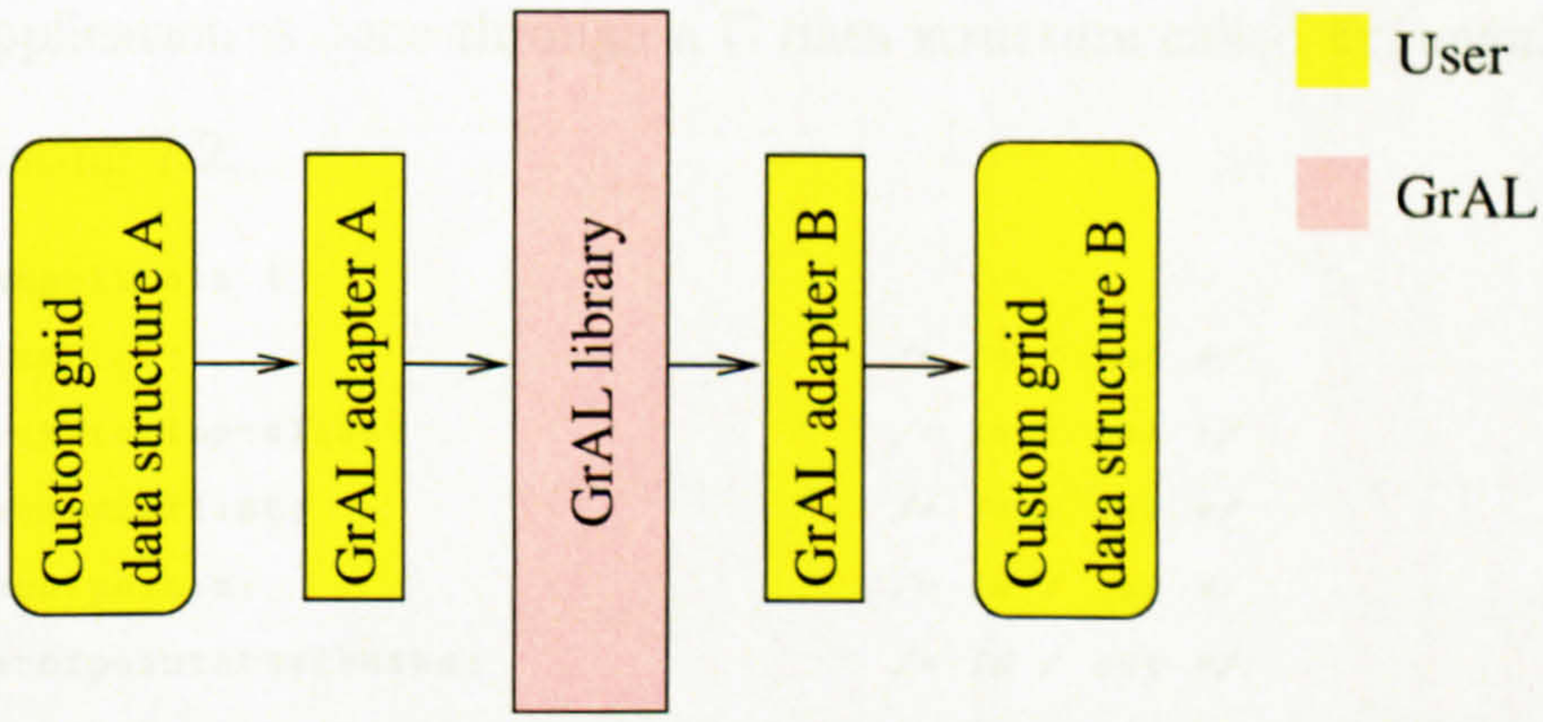


Figure 7.6: `GrAL` used to translate between custom data structures.

grid manipulation is done in an external library and `GrAL` only enables access to the user data structure.

A bit different situation is shown in Figure 7.6, where `GrAL` is used to translate from one user data structure to another. In this case `GrAL` generic implementation of copy operation is used.

7.6 GrAL interface to mesh generators

GrAL provides comprehensive set of grid manipulation routines, but except for regular Cartesian grids, it does not provide any mesh generation facilities. Thus the first two components added to GAGES were the GrAL interface to mesh generators.

7.6.1 GrAL interface to Triangle

Triangle [106] is well known and popular 2D Delaunay mesh generator written by R. Shewchuk. Triangle allows generation of exact Delaunay triangulations, constrained Delaunay triangulations, Voronoi diagrams, and quality conforming Delaunay triangulations. Triangle can be compiled into a stand alone program or can be used as a library. Communication between Triangle used as library and an external application is done through a C data structure called triangulateio and shown in listing 7.2.

```
struct triangulateio {
  REAL *pointlist;                                    /* In / out */
  REAL *pointattributelist;                           /* In / out */
  int *pointmarkerlist;                               /* In / out */
  int numberofpoints;                                 /* In / out */
  int numberofpointattributes;                        /* In / out */

  int *trianglelist;                                  /* In / out */
  REAL *triangleattributelist;                        /* In / out */
  REAL *trianglearealist;                             /* In only */
  int *neighborlist;                                  /* Out only */
  int numberoftriangles;                              /* In / out */
  int numberofcorners;                                /* In / out */
  int numberoftriangleattributes;                     /* In / out */

  int *segmentlist;                                   /* In / out */
  int *segmentmarkerlist;                             /* In / out */
  int numberofsegments;                               /* In / out */

  REAL *holelist;        /* In / pointer to array copied out */
  int numberofholes;                                  /* In / copied out */
```

```
REAL *regionlist;   /* In / pointer to array copied out */
int numberofregions;                    /* In / copied out */

int *edgelist;                                  /* Out only */
int *edgemarkerlist;  /* Not used with Voronoi diagram;
                                                  out only */
REAL *normlist;       /* Used only with Voronoi diagram;
                                                  out only */
  int numberofedges;                            /* Out only */
};
```

Listing 7.2: The data structure used to communicate with the `Triangle` mesh generator.

Though simple, this interface is not very convenient, especially having in mind future use in scripts or in interactive mode. Thus the goal in designing the `GrAL` interface to `Triangle` was to make this interface more user friendly and suitable for scripting.

Interface components

The `GrAL`'s `Triangle` interface is written in C++ and consists of four main classes:

1. `TriangleInput` – this class is in essence a container of elements constituting the input for `Triangle`, that is vertices, segments, holes and regions. All elements are kept in dynamically growable containers and `TriangleInput` provides a convenient interface for adding new elements and for iterating over existing ones.

2. `TriangleGenerator` – this class is a wrapper around the `triangulate` function from the `Triangle` library. It enables the generation of meshes from an input file or from instances of `TriangleInput`. Mesh generation parameters can be passed to `Triangle` by standard `Triangle` flags or can be set individually using respective `Set/Get` functions.

3. `TriangleAdapter` – this is the main class of the interface. It permits a `triangulateio` structure to masquerade as a `GrAL` compatible data struc-

ture by providing all sorts of required elements (handlers, iterators, etc.). This class deals only with topological data, while geometry is handled by `TriangleGeometryAdapter`.

4. `TriangleGeometryAdapter` – according to the GrAL design, the grids geometric aspects are separated from the topological aspects. `TriangleGeometryAdapter` provides an interface to vertex coordinates stored in the `triangulateio` structure and encapsulates other basic geometric aspects (e.g. dimensions, distance calculations, etc.).

Example

Listing 7.3 shows the usage of the GrAL Triangle adapter. To save some space the listing is stripped of all GrAL related header file inclusion directives.

```
// All GrAL header files skipped
using namespace GrAL;

#define REAL double
#include <triangle.h>

int main() {
  typedef Complex2D grid_type;
  typedef grid_types<grid_type> gt;

  triangle_generator::TriangleGenerator generator;
  triangle_generator::TriangleInput input;
  triangle_adapter::TriangleAdapter triangulation;

  input.addVertex(-1.0, -1.0);
  input.addVertex(1.0, -1.0);
  input.addVertex(1.0, 1.0);
  input.addVertex(-1.0, 1.0);

  input.addSegment(0,2);

  input.addRegion(-0.8,0.8,1.0,0.08);
  input.addRegion(0.8,-0.8,2.0,0.02);
```

```
generator.SetOptions("qpczA")
generator.Triangulate(input, triangulation);

Complex2D T;
stored_geometry_complex2D GeomT(T);
ConstructGrid(T, GeomT, triangulation, triangulation);

OstreamDX2DFmt Out("two_regions.out");
  ConstructGrid(Out,T,GeomT);
}
```

Listing 7.3: Example of using GrAL interface to Triangle mesh generator.

Mesh generated in this example fills square domain and is divided into two triangular regions with different mesh density.

7.6.2 GrAL interface to GRUMMP

GRUMMP is an acronym for the Generation and Refinement of Unstructured Mixed-Element Meshes in Parallel [109]. Though it is still a work in progress and does not live up to its acronym, it offers very interesting, efficient and robust generation tools. GRUMMP offers a set of C++ libraries supporting unstructured mesh generation and a set of executables including:

tri – a two dimensional triangular unstructured mesh generator for domains with curved boundaries with automatic control over cell size and grading [110],

tetra – a tetrahedral isotropic mesh generator for polyhedral domains with automatic control over cell size and grading [111].

Additionally GRUMMP package delivers tools for mesh improvement (`meshopt2d` and `meshopt3d`), mesh coarsening (`coarsen2d` and `coarsen3d`), and interpolation of scattered data (`scat2d` and `scat3d`).

Interface components

In the work presented in this thesis an interface to the two dimensional mesh generator (`tri`) has been built. This interface consists of three main classes:

1. `Boundary2D` – this class can be treated as a front-end to the GRUMMP native class Bdry2D. `Boundary2D` is essentially a container class for objects holding data for boundary patches. GRUMMP and at the same time `Boundary2D` provides five kinds of boundary patches:

   - `Polyline`, for polygonal boundaries,
   - `Circle`, for circles,
   - `Arc`, for circular arcs,
   - `Bezier`, for Bêzier curves,
   - `Spline`, for cubic interpolated splines.

   Using these patches it is possible to describe non-polygonal multiple-subdomain meshing regions. All boundary patches are kept in `Boundary2D` in dynamically growable containers, and `Boundary2D` offers a convenient interface for adding, removing, iterating over them and querying their properties.

2. `TriGenerator` – this class is basically a wrapper over GRUMMP's `tri` generator. It permits the same meshing parameters as `tri` does. `TriGenerator` works with the two other interface classes – `Boundary2D` and `Mesh2DAdapter`. Input to the mesh generator can come either from the GRUMMP (.bdry) file or from the `Boundary2D` object.

3. `Mesh2DAdapter` – this is the core class of the GrAL interface. It masquerades as GRUMMP's native class `Mesh2D` by providing all required interface (methods, grid element classes, iterator classes) so it is possible to use it with `GrAL`'s algorithms.

Example

The use of the GRUMMP interface is shown in listing 7.4 and the generated mesh in Figure 7.7. Please note, that in order to save some space all `#include` directives have been removed from the listing.

```
int main() {
  using namespace GrAL;
  using namespace grummp_generator;

  typedef Complex2D grid_type;
  typedef grid_types<grid_type> gt;

  Mesh2DAdapter gadapt;
  TriGenerator generator;
  Boundary2D boundary;

  boundary.addPoint(-0.2,0);     boundary.addPoint(0.5,0);
  boundary.addPoint(1,0.5);      boundary.addPoint(1,1.0);
  boundary.addPoint(-0.2,1);     boundary.addPoint(0.5,0.5);
  boundary.addPoint(0.72,0);     boundary.addPoint(1.0,0.28);

  Boundary2D::Polyline p(boundary, Boundary2D::REGION, 1,
                                   Boundary2D::BOUNDARY, 1);
  p.addPoint(2); p.addPoint(3);
  p.addPoint(4); p.addPoint(0);
  p.addPoint(1);

  boundary.addPolyline(p);

  Boundary2D::Circle c(boundary, Boundary2D::BOUNDARY, 2,
                                 Boundary2D::REGION, 1, 4, 0.3);
  boundary.addCircle(c);

  Boundary2D::Bezier bez(boundary, Boundary2D.REGION, 1,
                                   Boundary2D.BOUNDARY, 1, 1,2,6,7);

  boundary.addBezier(bez);
  generator.triangulate(boundary, "", gadapt);

  Complex2D T;
  stored_geometry_complex2D GeomT(T);
  ConstructGrid(T, GeomT, gadapt, gadapt);
```

```
  OstreamDX2DFmt Out("circhole.dx");
  ConstructGrid(Out,T,GeomT);

  return 0;
}
```

Listing 7.4: Example of using the GrAL interface to the GRUMMP mesh generator.

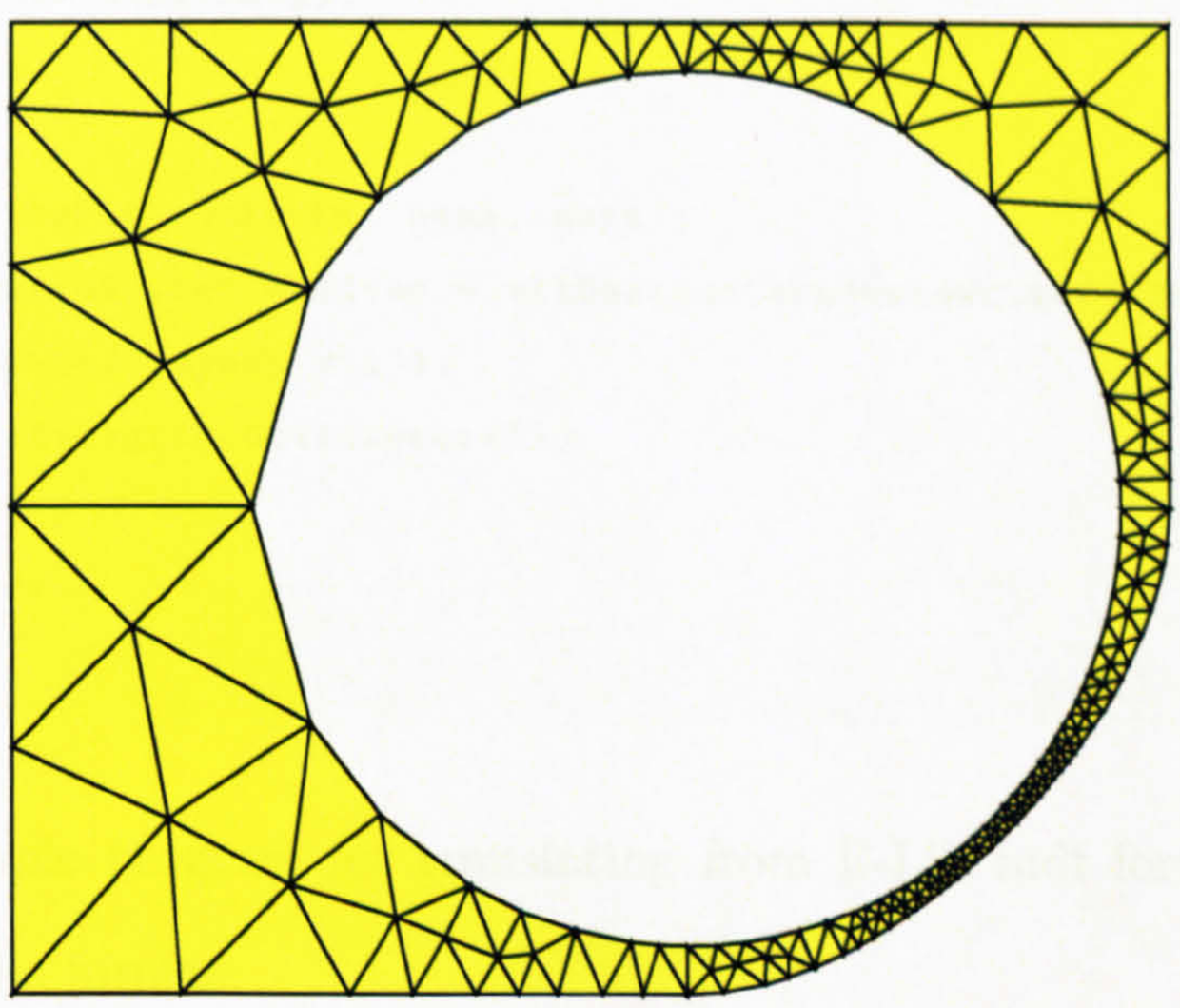


Figure 7.7: Mesh generated by program from listing 7.4

7.6.3 Data structure adapters and output filters

In this section a concrete examples of the idea of GrAL adapters discussed in section 7.5.2 are given. Listing 7.5 shows a program built according to the scheme shown in Figure 7.6 which translates between the MESH data structure from E-Lib library and vtkUnstructuredGrid data structure from the VTK library. Please note as a result of space restrictions all #include directives has been removed from the listing.

```
int main(int argc, char *argv[]) {
  using namespace GrAL;
  using namespace GrAL::vtk_ugrid;
  typedef Complex2D grid_type;
  typedef grid_types<grid_type> gt;
```

```
if (argc < 2 ) {
  std::cerr << "Usage: " << argv[0] << " mdf_basename\n";
  exit(1);
}

e_lib_adapter::MESHAdapter2D mesh(argv[1]);

vtkUnstructuredGrid *g;
  UGridVTKAdapter<2> vtkgrid(g);
  vtkgrid.clear();

  ConstructGrid(vtkgrid, vtkgrid, mesh, mesh);
  vtkUnstructuredGridWriter *writer = vtkUnstructuredGridWriter::New();
  writer->SetFileName("mayavi.vtk");
  writer->SetInput(vtkgrid.GetAdaptee());
  writer->Write();
  writer->Delete();
  return 0;
}
```

Listing 7.5: Sample program for translating from E-Lib mdf format to VTK unstructured grid file format.

Figures 7.8 and 7.9 show the visualisation of data produced by program from listing 7.5 after it was saved in its respective file format. To visualise `E-Lib's MESH` data structure the program `eplx` [5] has been used and to visualise `VTK's vtkUnstructuredGrid` the program `mayavi` [136] has been used.

The `GrAL` adapters showed in the previous example allow in-core translation between data structures and use of them with `GrAL` based algorithms. The example below shows another kind of interface – an output filter to `OpenDX`[5] native data format. In this case no external library has to be used, though it is possible to build an alternative implementation where the `GrAL` based grid is translated in core to an `OpenDX` data structure and only then saved to a file using the `OpenDX` API.

```
int main(int argc, char *argv[]) {
  if (argc < 2 ) {
    std::cerr << "Usage: " << argv[0] << " mdf_basename\n";
```

[5]Open Source version of IBM Data Explorer.

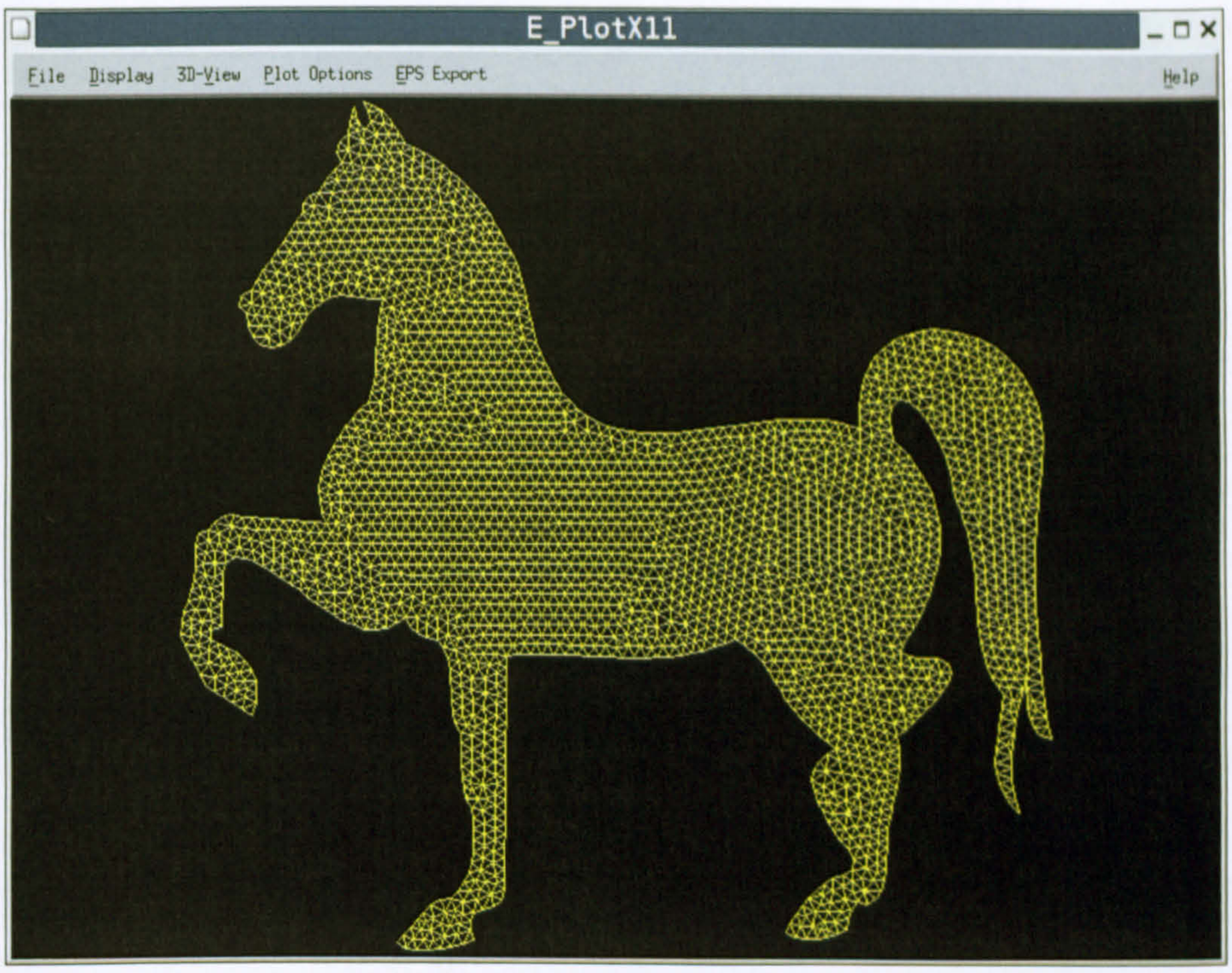


Figure 7.8: Horse mesh visualised using the `eplx` [5] program.

```
  exit(1);
}

e_lib_adapter::MESHAdapter2D mesh(argv[1]);

OstreamDX2DFmt Out("horse.dx");
ConstructGrid(Out,mesh,mesh);

return 0;
}
```

Listing 7.6: Sample program for translating from E-Lib mdf format to OpenDX native format.

With few a simple modifications the above program is a perfect example of tool forming command line tool to the GAGES layer.

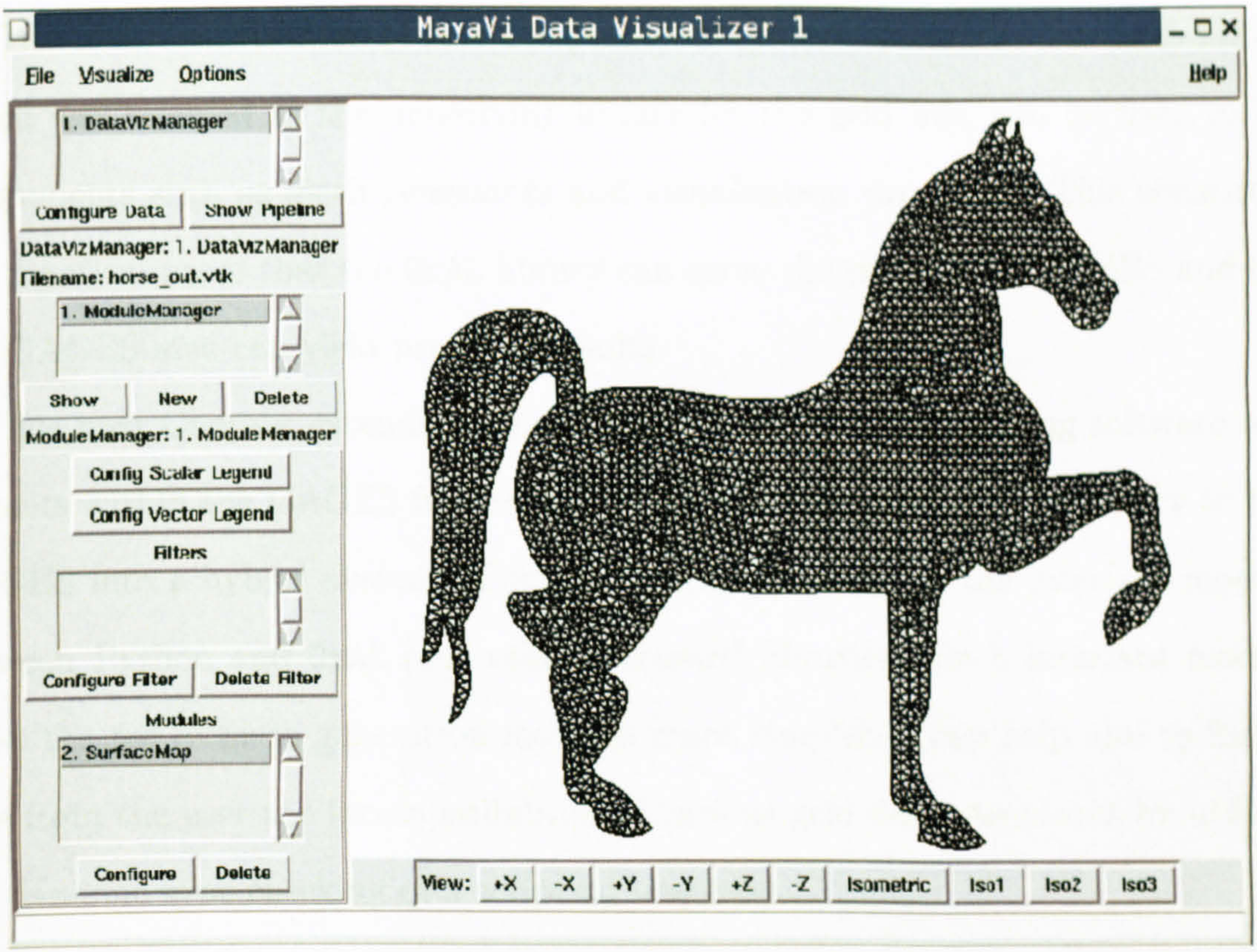


Figure 7.9: Horse mesh translated to VTK format and visualised using the `mayavi` program.

7.7 Concluding remarks

The discussion presented in this Chapter concentrated on the reuse of computational grids which are one of the most common data structures in the scientific computing. Much of this discussion was devoted to the grid data structure diversity which is the main problem to solve when integrating grid based software components. Understanding the source of this diversity is an important step towards designing a generic mechanism for the exchange of grid data. This Chapter presented the requirements for such mechanism and discussed two categories of it – one based on in-core data transfer and another based on off-core data transfer. It was shown that providing only bare specification of exchange data formats is less advantageous to in-core data transfer. Thus the rest of the discussion was concentrated only on in-core data exchange.

The main contribution of this Chapter is the presentation how the GrAL library, which was selected as the underlying library for the grid bus, can be used to link components such as mesh generators and visualisation programs. This constitutes the practical proof that the GrAL library can serve the purpose of GAGES and that the GAGES idea can yield practical results.

The next Chapter extends the presented discussion about linking software components within the GAGES framework. It presents the next step necessary to turn GAGES into a hybrid system. This step is the provision of the interface modules between Python and GrAL plus other discussed libraries. Such interface modules make the use of mesh generation modules much simpler. They help also to further hide from the user the incompatibilities of various grid data structures by utilising the dynamic type checking of a scripting language.

Chapter 8

Gluing GAGES components with scripting languages

All the advantages of the generic approach described in section 3.5 come however at some cost. GrAL uses a fairly sophisticated design patterns and modern C++ programming techniques, thus its steep learning curve may be prohibitive for a casual user. Besides, GrAL cannot be easily used in the area of throw-away programming, rapid prototyping or computational steering. In the recent years these areas have been dominated by the scripting languages such as Python, Ruby, Tcl or Perl. Among them Python deserves special attention as it already contains a rich set of scientific tools and can be easily mixed with C/C++, FORTRAN , or Java. Mixed language programming – scripting languages for their flexibility and compiled languages for their efficiency is now a well established programming technique for developing advanced scientific simulation codes.

Thus the practical question arises if and how GrAL can be brought to a scripted environment, or more specifically, if it is possible to provide a Python interface to GrAL. This Chapter shows that it is both possible and advantageous to construct Python interface to GrAL and other libraries, as this results in an extremely flexible environment for grid manipulations. This Chapter presents a detailed account on the study of the feasibility of providing a scripting interface to the base GRAL

components. It shows that it is possible to turn the whole GAGES architecture into a hybrid system as discussed in section 4.3. The place of the presented material in the whole discussion about the GAGES architecture is shown in Figure 8.1.

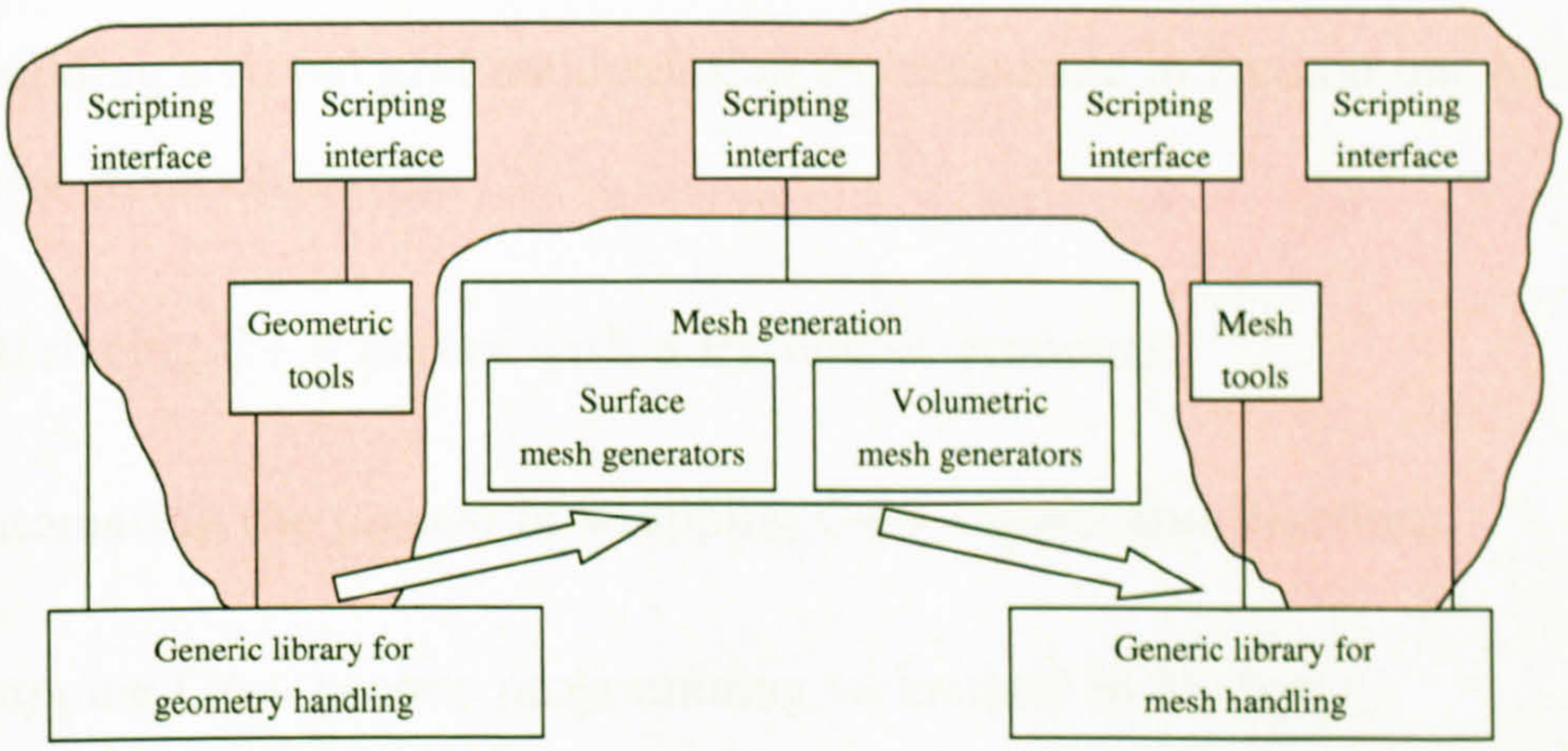


Figure 8.1: The place of the material presented in this Chapter in the whole discussion about the GAGES architecture.

8.1 Integrating Python and C/C++.

One attractive aspect about Python (as well as about other modern scripting languages) is that they can be easily interfaced with the C language. Python comes with an extensive and well documented C Application Programmer Interface (API). Python's standard library can be extended using this API by various C/C++ libraries, and Python scripts can be embedded in C/C++ applications for instance as configuration files.

A problem with direct use of the Python C API is that it forces the programmer to write a lot of repetitive code. Each function, method, or global variable requires a wrapper function. Even with moderately sized libraries writing and maintaining such code is costly and error prone.

Another problem with Python API is that it does not provide enough support for C++. Though in principle it is possible to interface the C++ code using the

Python C API, it comes however at an additional cost.

8.1.1 Scripting language interface

When building scripted grid manipulation environments in Python one has to overcome three main obstacles:

- interfacing C++ library with a Python environment,
- automating the process of wrapping C++ classes and functions,
- mapping C++ generic programming techniques to Python.

8.1.2 Automatic integration of C++ and Python

The difficulties mentioned above can be overcome by using one of several packages for automatic generation of C++/Python interfaces. The most popular packages include:

- SWIG
- Boost.Python
- SIP
- Weave
- Pyrex

Taking into account that wrapping the `GrAL` library requires really good support for advanced C++ features like templates, namespaces, nested classes, etc., only the two first packages, i.e. SWIG [119] and Boost.Python seem to meet the requirements. As the author had more experience with SWIG, this package was chosen for wrapping the `GrAL` library.

8.2 PyGrAL

The biggest problem in interfacing the GrAL library with Python is that GrAL code relies heavily on C++ templates and elements of C++ metaprogramming. The problem with templates is that there is no equivalent Python feature on which template declarations can be mapped. C++ template declarations are handled at compile time and they do not define any runnable object-code which can be wrapped.

The way SWIG handles templates is that they must be instantiated and only concrete template instances can be wrapped by SWIG. This way GrAL code wrapped in Python looses its generic nature. It does not however pose a big problem if template instances can be automatically wrapped for new types.

Handling simple template declarations is rather straightforward. In GrAL however several sophisticated template constructs are used, for instance template partial specialisation, template parameters which are themselves templates, recursive templates. For some constructs SWIG provides restricted support but others must be cleverly resolved by manual modifications.

Though SWIG automates the wrapping of the C++ code to a great extent, using it for such a complex library like GrAL presents several difficulties. The next section discusses a couple of useful techniques, but also points to some problems resulting in either a clumsy Python interface or a wrapper code that requires manual adjustments.

8.2.1 Organisation of GrAL/Pyton interface

The GrAL library is partitioned into logical packages which in turn are divided into modules. The high level view of GrAL packages is shown below:

```
GrAL
 |---- Basic packages
 |     |-- GrAL-Base package.
```

```
|     |-- GrAL-Testing package.
|---- General unstructured grid data structures
|    |-- GrAL-Complex2D package
|    |-- GrAL-Complex3D package
|    |-- GrAL-ComplexND package
|---- Triangulation data structures
|    |-- GrAL-Triang2D package
|    |-- GrAL-Triang3D package
|---- Cartesian grids
|    |-- GrAL-Cartesian2D package
|    |-- GrAL-Cartesian3D package
|    |-- GrAL-CartesianND package
|---- Algorithmic packages
|    |-- GrAL-Partitioning package
|    |-- GrAL-Geometry package
|    |-- GrAL-Measurement package
|---- Data structures for distributed and hierarchical grids
|    |-- GrAL-Distributed package
|    |-- GrAL-Hierarchical package
|---- I/O to various file formats
|    |-- GrAL-GMV-IO package
|    |-- GrAL-Geomview-IO package
```

The generic nature of the GrAL library allows minimal dependencies to be maintained between packages. For instance the Partitioning package provides a partitioning algorithm regardless of the type of a grid data structure it works on.

In Python however such a packages organisation is impossible due to the fact that all template classes and functions need to be explicitly instantiated. For example, there must be a separate instance of the partitioning function for Cartesian2D grids,

Complex2D grids and so on. Thus most of the Python's GrAL interface is organised around particular grid data structures. Each grid data structure becomes a root of a tree containing classes and functions specialised for this structure. The packages which do not depend on a grid data structure are organised in a manner similar to GrAL. Below a part of the Python's GrAL package tree is shown.

```
Packages/
|-- Cartesian2D
|    |-- Algorithms
|    |-- Base
|    |-- Geometries
|    |-- IO
|    |-- Partitioning
|    |-- cartesian2d.py
|    |-- grid_functions.py
|    |-- mapped_geometry.py
|    |-- stored_geometry.py
|-- Complex2D
|    |-- Geometries
|    |-- IO
|    |-- Iterators
|    |-- Partitioning
|    |-- Subranges
|    |-- complex2d.py
|    |-- construct.py
|    |-- grid_functions.py
|    |-- stored_geometry.py
|    |-- test_grids.py
|-- Container
```

```
|   |-- tuple.py
|-- Geometry
|   |-- box.py
|   |-- coords.py
|-- IO
|   |-- complex2d.py
|   |-- dx2d.py
|-- Triang2D
|   |-- Algorithms
|   |-- Base
|   |-- Distributed
|   |-- Geometries
|   |-- IO
|   |-- Iterators
|   |-- Partitioning
|   |-- Subranges
|   |-- Views
|-- Utils
|-- View
```

8.2.2 GrAL iterators

One of the fundamental concepts in GrAL are iterators. GrAL iterators allow collections of grid elements (vertices, edges, faces, cells) to be treated as linear sequences. Typical use of GrAL iterators is illustrated by the program shown in listing 8.1.

```
1  #include "Gral/Base/element-numbering.h"
2  #include "Gral/Grids/Cartesian2D/all.h"
3
4  int main() {
5    using namespace GrAL;
6    namespace c2d = cartesian2d;
```

```
7
8     // typedef several types to simplify declarations
9     typedef c2d::CartesianGrid2D  grid_type;
10    typedef typename grid_type::Vertex Vertex;
11    typedef typename grid_type::Cell Cell;
12    typedef typename grid_type::CellIterator CellIterator;
13    typedef typename grid_type::VertexOnCellIterator VertexIterator;
14
15    // Cartesian grid with 3 vertices
16    // in X direction  and 3 vertices
17    // in Y direction.
18    grid_type grid(3,3);
19
20    for(CellIterator c(grid); !c.IsDone(); ++c) {
21      for(VertexIterator v(*c); !v.IsDone(); ++v) {
22            std::cout << v.handle() << " ";
23      }
24      std::cout << "\n";
25   }
26 }
```

Listing 8.1: Example of a simple GrAL based program.

This program will print for each cell the indices of vertices incident to the cell. GrAL iterators are basically classes which provide dereference `operator*()`, pre-increment `operator++()` and method `IsDone()` to test for the iteration end. In Python the above code takes the form:

```
1 from sys import stdout
2 from GrAL.Cartesian2D.cartesian2d import Cartesian2D
3
4 grid = Cartesian2D(3,3) # create grid
5
6 for cell in grid.iterCells():
7   for vertex in cell.iterVertices():
8     stdout.write(" %d" % vertex.handle())
9   stdout.write("\n")
```

Listing 8.2: Python equivalent of program shown in listing 8.1.

Though not visible at first sight, the code above also uses iterators which in Python are a fundamental part of the language. The iterator is implicitly used by the `for`

statement. Functions `iterCells()` and `iterVertices()` return iterator objects. Iterator is basically an object which provides two methods: `__iter__()` which creates an iterator and `next()` which returns the next iterated value. The `next()` method may raise a `StopIteration` exception when iteration has reached a logical end.

A simple example of an iterator class in Python is shown in listing 8.3

```
1   class BackwardCounter:
2     def __init__(self, stop):
3       self.stop = stop
4     def __iter__(self):
5       return self
6     def next(self):
7       if self.stop < 1:
8         raise StopIteration
9       else:
10        s = self.stop
11        self.stop -=1
12        return s
13
14  for num in BackwardCounter(10):
15    print num
```

Listing 8.3: Sample iterator class in Python.

Advancing the iterator produces a sequence of decreasing integer values until 0 is reached.

To wrap `GrAL` iterators in a way, so that they mimic the behaviour of Python iterators as closely as possible, requires augmenting the iterator class declaration with additional SWIG directives. The directives take care of adding methods `__iter__()` and `next()` to the wrapped class (%extend directive) as well as object owning issues (%newobject directive). For convenience the SWIG directives are wrapped as the macro `EXTEND_VALUER_ITERATOR` which takes as arguments the name of the wrapped class and the name of the iterator value type. The complete code for this macro is shown in listing 8.4.

```
1  %define EXTEND_VALUE_ITERATOR(IterClass, ValueType)
2  %newobject IterClass::next();
3  %newobject IterClass::__iter__;
4  %ignore IterClass::operator++;
```

```
5  %extend IterClass {
6   inline ValueType* next() {
7          if (self->IsDone() == true) {
8     PyErr_SetObject(PyExc_StopIteration, Py_None);
9                 return NULL;
10    }
11    ValueType* result = new ValueType(self->operator*());
12         ++(*self);
13    return result;
14  };
15  inline IterClass *__iter__() {
16   return new IterClass(*self);
17  };
18  /* substitute for operator++ */
19  inline bool advance() {
20         if (self->IsDone() == true) {
21    return false;
22   }
23   ++(*self);
24         if (self->IsDone() == true) {
25    return false;
26    }
27    return true;
28   };
29  }
30  %enddef
```

Listing 8.4: SWIG macro to automate wrapping iterator classes.

8.2.3 Grid functions

Another non-trivial issue when wrapping the GrAL library is support for grid functions. Grid function is a template class which enables the storing of arbitrary data on grid elements like vertices, edges, cells, etc. In other words it provides mapping from grid elements of some fixed type (vertices, edges, cells) to values of some type T. The usage of GrAL grid functions is shown in listing 8.5.

```
1  #include "Gral/Base/element-numbering.h"
2  #include "Gral/Grids/Cartesian2D/all.h"
3
4  int main() {
```

```
5     using namespace GrAL;
6     namespace c2d = cartesian2d;
7
8     // typedef several types to simplify
9     // declarations
10    typedef c2d::CartesianGrid2D  grid_type;
11    typedef typename G::Vertex Vertex;
12    typedef typename G::Cell Cell;
13    typedef typename G::CellIterator CellIterator;
14    typedef typename G::VertexIterator VertexIterator;
15    typedef typename G::VertexOnCellIterator VertexOnCellIterator;
16
17    // Cartesian grid with 3 vertices
18    // in X direction and 3 vertices
19    // in Y direction.
20    grid_type grid(3,3);
21
22    grid_function<Cell, double> gfc(grid);
23    grid_function<Vertex, int>  gfv(grid);
24
25    int i=1;
26    for(VertexIterator v(grid);
27                      ! v.IsDone(); ++v) {
28      gfv[*v] = i++;
29    }
30
31    for(CellIterator c(grid); ! c.IsDone(); ++c) {
32      double accumulator = 0;
33      for(VertexOnCellIterator v(*c); !v.IsDone(); ++v) {
34           accumulator += gfv[*v];
35           std::cout << gfv[*v] << " ";
36      }
37
38      gfc[*c] = accumulator/(*c).NumOfVertices();
39      std::cout << " cell average = " << gfc[*c] << "\n";
40    }
```

Listing 8.5: Example of using grid functions in GrAL.

The above program uses two grid functions, one defined over vertices with values of integer type and one defined over cells with values of double precision type. The first loop assigns values to each vertex and the second loop calculates, stores and

displays the cell value being the average of the values of the vertices adjacent to the cell.

The above code rewritten in Python takes the form shown in listing 8.6.

```
1  from sys import stdout
2  from GrAL.Cartesian2D.cartesian2d import Cartesian2D
3  from GrAL.Cartesian2D.grid_functions import grid_function_on_vertices
4  from GrAL.Cartesian2D.grid_functions import grid_function_on_cells
5
6  grid = Cartesian2D(3,3)
7  gfv = grid_function_on_vertices(grid)
8  gfc = grid_function_on_cells(grid)
9
10 i=1
11 for vertex in grid.iterVertices():
12     gfv[vertex] = i
13     i+=1
14
15 for cell in grid.iterCells():
16     accumulator=0
17     for vertex in cell.iterVertices():
18         stdout.write(" %d" % gfv[vertex])
19         accumulator += gfv[vertex]
20     gvc[cell] = accumulator / cell.NumOfVertices()
21     stdout.write(" cell average= %f\n" % gvc[cell])
```

Listing 8.6: Python equivalent of program from listing 8.5.

GrAL grid functions are parameterised with two types, argument type and value types. According to what was said about wrapping C++ templates both types must be provided *a priori*. Argument types pose no problem as they can only be either a vertex, edge, face, facet or cell type. Thus for each grid type one has `grid_function_on_vertices`, `grid_function_on_edges` and so on. As the function value type is concerned, then instead of providing instances for the most popular types like `int`, `double`, etc. the value type is selected to be of `PyObject*` pointer. Hence Python wrappers for GrAL grid functions can store any Python object. A side effect of this design is that now grid functions are heterogeneous if the value type is concerned. There is however a prices to pay for this flexibility – grid functions

created at a Python level cannot be directly used at the C++ level and vice versa.

Grid functions provide the [ ] operator which gives them array-like semantic as well as two methods: `iterkeys()` and `itervalues()`. The first permits the iteration over grid function arguments and the second allows iteration over grid function values. Iteration over grid function values requires slightly different handling as the values are now `PyObject` pointers. Because Python objects are reference counted the wrapper code must take it into account. As in the case with "ordinary" iterators all SWIG directives necessary to wrap `PyObject*` valued iterators are provided by a macro EXTEND_NATIVE_ITERAOR. The code of which is shown in listing 8.7.

```
1  %define EXTEND_NATIVE_ITERATOR(IterClass)
2  %newobject IterClass::__iter__;
3  %ignore IterClass::operator++;
4  %extend IterClass {
5    inline PyObject* next() {
6      if (self->IsDone() == true) {
7        PyErr_SetObject(PyExc_StopIteration, Py_None);
8        return NULL;
9      }
10     PyObject* retval = self->operator*();
11     ++(*self);
12     Py_INCREF(retval);
13     return retval;
14   };
15   inline IterClass *__iter__() {
16     return new IterClass(*self);
17   };
18 }
```

Listing 8.7: SWIG macro to support wrapping grid functions.

8.2.4 Iteration over grid boundary elements

The script in listing 8.8 shows how one can read a mesh and iterate over boundary edges. The script uses class `IstreamComplex2DFmt` to read a 2D mesh saved in a very simple text file. Then for each boundary edge the handles of start (`edge.V1()`) and end (`edge.V2()`) vertices are printed.

```
1  from sys import argv
2
3  # import necessary classes and functions
4  from GrAL.IO.complex2d  import IstreamComplex2DFmt
5  from GrAL.Complex2D.complex2d  import Complex2D
6  from GrAL.Complex2D.stored_geometry import stored_geometry_complex2D
7  from GrAL.Complex2D.construct import ConstructGrid
8
9  # create input adapter
10 # the first argument is file name
11 # the second argument is numbering base
12 input = IstreamComplex2DFmt(argv[1], int(argv[2]))
13
14 # create grid and geometry objects
15 grid = Complex2D()
16 geom = stored_geometry_complex2D(grid)
17
18 # fill the grid and geometry with data
19 # read form input
20 ConstructGrid(grid, geom, input, input)
21
22 for edge in grid.iterBoundaryEdges():
23     print "Edge %d %d" % (edge.V1().handle(), edge.V2().handle())
```

Listing 8.8: Example of iteration over boundary edges.

8.3 Libplot, LPlotter and ONPlot

Though not strictly a GAGES component the `libplot`, library is mentioned here as it was a base for providing 2D visualisation tools for grids and geometries. This library is a part of GNU Plotutils [134] package and provides a C based API for drawing two-dimensional vector graphics. An important aspect of `libplot` is that the same code can be used to produce graphics in many file formats as well as double-buffered animations for the X Window System. The ability to produce xfig and eps output makes `libplot` an excellent drawing tool for LaTeX documents.

The greatest disadvantage of `libplot` is that all plotting coordinates are specified in the users space. This makes it difficult to position elements like annotation,

title and legends independently of the plotting contents. In order to mitigate this disadvantage the `LPlotter` C++ library was built on top of `libplot` API, for the purpose of this thesis. The `LPlotter` library provides the `LPlotter` class. Objects of this class store extreme coordinates of both user space and viewport space. Hence it is possible to map positions and dimensions from viewport space to user space, and vice versa, at any time.

The `LPlotter` provides a set of generic plotting primitives such as lines, circles, boxes, ellipses, points, and Bezier curves. They can be used to construct graphical representations of almost any 2D object. However, to avoid code duplication another small utility library `ONPlot` was created on top of `LPlotter`. The `ONPlot` library is intended to help visualising OpenNurbs objects. Though OpenNurbs objects are in principle three-dimensional and can be visualised with OpenGL, many explanatory drawings, especially for curves, can be done in two dimensions and with `ONPlot` they can be saved in small scalable vector formats (eps, fig, svg), suitable for publication.

All three libraries `libplot`, `LPlotter` and `ONPlot` have very classical interfaces and it was a matter of a couple of hours work to build SWIG Python interfaces to them.

The interfaces to `LPlotter` and `ONPlot` are shown in appendix B and C, respectively. From the time perspective it is important to note that the `LPlotter` interface is an example of a bad design. The author could not decide if to base the interface on inheritance or delegation, changed the approach several times and additionally tried to add to many features to the library. This became especially visible after creating the Python interface to `LPlotter` as this interface does not feel very intuitive, especially in the case of handling plotting styles. Despite these disadvantages combining `libplot` with Python yielded a useful tool – an example of this is the fact that almost all two dimensional plots in this dissertation were made with the `LPlotter` library.

8.4 PyGrAL viewing tools

The GrAL library does not provide itself with any visualisation services, though version 1.0 will provide a generic graphics-device module. The grids manipulated by GrAL can be of course visualised by saving them in VTK, OpenDX, GeomView, or GMV file formats and using an appropriate viewing tool, but often a simple programming interface is required to control grid plotting. Plotting tools could either be built in a C++ layer or in a scripting language layer, and it was decided that it would be easier to do it in Python. Beside the mentioned interfaces to `libplot`, `LPlotter` and `ONPlot` libraries, Python provides interfaces for `OpenGL`, `PLplot`, Postscript and other plotting solutions. Thanks to them it is possible to build a single grid plotting package with multiple outputs based on the mentioned interfaces. Such a package has been indeed built for the purpose of this thesis and it is a part of PyGrAL. The PyGrAL viewing package consists of the following modules:

- `View2DLibplot` – a module for plotting 2-dimensional GrAL compatible grids.
- `ViewPartitions` – a module for plotting grid partitioning on the basis of GrAL's `Partitioning` compatible objects. Figure 8.6 was created using this module.
- `TriangleInputViewer` – a module for plotting `triangle` generator input data stored in `TriangleInput` class from `triangle` to GrAL adapter. Figure 9.5 was created using this module.

```
#!/usr/bin/env python
import sys

from GrAL.IO.complex2d import IstreamComplex2DFmt
from GrAL.Complex2D.complex2d import Complex2D
from GrAL.Complex2D.stored_geometry import stored_geometry_complex2D
from GrAL.Complex2D.construct import ConstructGrid
from GrAL.View.View2DLibplot import View2DLibplot

# The first argument to the script is data file name
```

```
# and the second is vertices numbering offeset (0 or 1)
input = IstreamComplex2DFmt(sys.argv[1], int(sys.argv[2]))

grid = Complex2D()
geom = stored_geometry_complex2D(grid)

ConstructGrid(grid, geom, input, input)

file = open("grid.fig", "w")
viewer = View2DLibplot(type="fig", outfile=file)
viewer.setViewForGeometry(geom, border=20, radius=4)

viewer.drawCells(geom)

viewer.savestate()
viewer.flinewidth(0.03)
edge = grid.FirstEdge().value()
viewer.drawEdge(edge, geom, color="magenta")
viewer.restorestate()

viewer.drawVertices(geom, color="blue")
viewer.ffontsize(0.08)
viewer.drawVertexIndices(geom)
viewer.ffontsize(0.1)
viewer.pencolorname("red")
viewer.drawCellIndices(geom)
```

Listing 8.9: Example of grid 2D grid visualisation using `GrAL.View.View2DLibplot` module.

```
8 3
0.0 0.0
1.0 0.0
1.0 1.0
0.0 1.0
0.5 0.5
0.5 1.5
-0.5 0.75
-0.5 0.25

5  1 2 5 3 4
3  3 4 6
```

```
4  4 1 8 7
```

Listing 8.10: Grid data for example 8.9

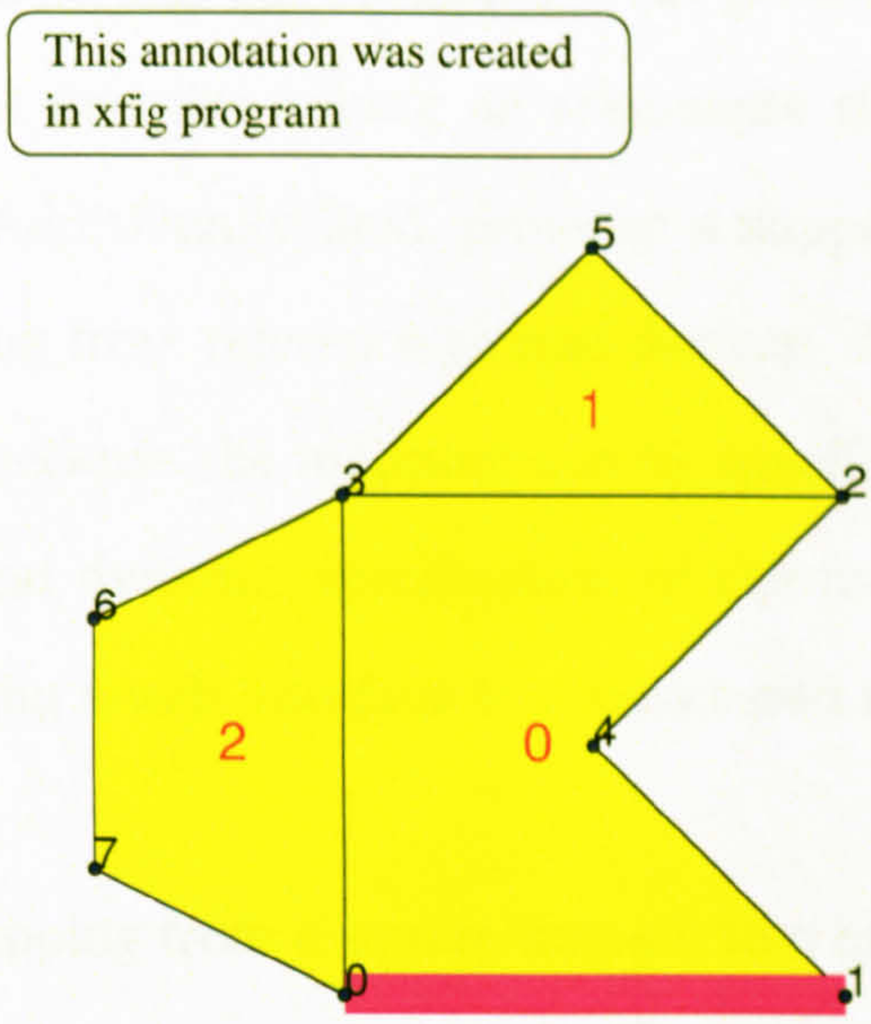


Figure 8.2: Figure produced using the code shown in listing 8.9, saved in fig format and modified by `xfig` program.

The simplicity of producing pictures with these modules is shown in listing 8.9. In this program an unstructured mesh is produced on the basis of data stored in the file shown in listing 8.10.

8.5 Structured grid generation with PyGrAL

One of the simplest techniques for generating structured grids is to generate it in a simple, usually square reference domain, and then to map it onto a real geometric domain in such way, that the reference domain boundaries are mapped exactly onto real domain boundaries. Such grids are called boundary fitted grids. The mapping can be done by finding explicit algebraic formulas or by solving appropriate partial differential equations. The advantage of the algebraic approach is its simplicity and speed, however the difficulty may lay in finding an appropriate mapping. This is especially hard for domains with holes or with very distorted boundaries.

Nevertheless, assuming that one has an appropriate mapping, generation of boundary fitted grids in GrAL is relatively easy. GrAL classes for structured grids Cartesian2D, Cartesian3D and CartesianND<> all generate grids in square (cube, or hyper-cube) reference domains taking as arguments the grid resolutions along particular dimensions. Additionally GrAL provides a mapped_geometry class which encapsulates the mapping from reference to real domain. In PyGrAL using these facilities is even simpler, because the mapping can by specified as a Python function. This allows a flexible and dynamic specification of the mapping, and that feature was utilised when building a web interface to a structured mesh generator described in section A.3.2.

Let us consider a mapping from a square domain to a quarter of an annulus with internal radius of 1 and external radius of $1 + \sqrt{2}$. Such a mapping is given by the following equations [95]:

$$x = 4st(1-s)(1-t) + \ (1 + t * \sqrt{2}) \cos\left(\frac{\pi}{2}s\right) \tag{8.1}$$

$$y = (1 + t\sqrt{2}) \sin\left(\frac{\pi}{2}s\right) \tag{8.2}$$

The code in listing 8.11 shows how to encapsulate such a mapping in a Python class. The use of this class to effectively generate a grid shown in Figure 8.3 is given in listing 8.13.

```
import math
class Anulus:
    __sqrt2 = math.sqrt(2.0)
    __PI_2 = math.pi/2.0

    def map(self, s, t):
        x = 4*s*t*(1-s)*(1-t) + \
            (1+t*self.__sqrt2) * math.cos(self.__PI_2 * s)
        y = (1+t*self.__sqrt2)*math.sin(self.__PI_2 * s)
        return (x,y)
```

Listing 8.11: Class encapsulating mapping from a reference square to quarter of annulus.

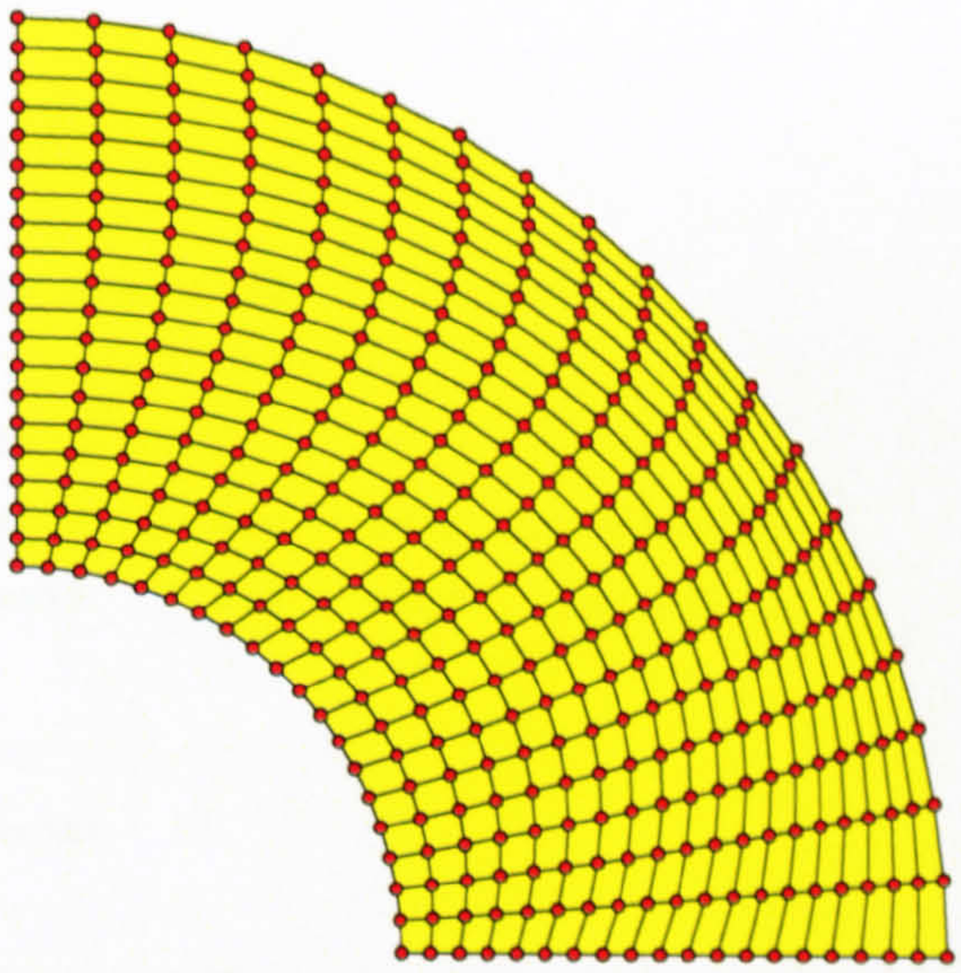


Figure 8.3: Grid generated through algebraic map shown in listing 8.11.

In some cases it is not easy to find explicit mapping formulas. Such cases can be helped by building the mapping as a linear interpolation of the mappings describing the domain boundaries. Since the boundary values are given at an infinite number of points such mappings are called transfinite interpolations (TFI). The derivations of the TFI formulas for a square and cube can be found in [95] and here we only show the code for class `BilinearBlendedSquare` which encapsulates a TFI mapping. The constructor for this class takes four arguments which are arbitrary functions describing region boundaries (the only requirement is that the functions are geometrically continuous at domain corners). The use of this class for generating the grid shown in Figure 8.4 is shown in listing 8.13.

```
#!/usr/bin/env python
class BilinearBlendedSquare(object):
    def __init__(self, south=None, east=None, north=None, west=None):
        self.set_south(south)
        self.set_north(north)
        self.set_west(west)
        self.set_east(east)

    #identity mapping used when no user mapping is given
    def id(sef, s,t):
        return (s,t)
```

```
def set_south(self, map):
    if map is None:
        self.__south = self.id
    else:
        self.__south = map
    self.__c_swx, self.__c_swy = self.__south(0,0)
    self.__c_sex, self.__c_sey = self.__south(1,0)

def get_south(self):
    return self.__south

# the code for setting north, east and west mapping is analogous
# and has been skipped

south = property(get_south, set_south)
north = property(get_north, set_north)
east = property(get_east, set_east)
west = property(get_west, set_west)

def map(self, s, t):
    x_s, y_s = self.__south(s,0)
    x_n, y_n = self.__north(s,1)
    x_w, y_w = self.__west(0, t)
    x_e, y_e = self.__east(1, t)

    cx = - (1-s)*(1-t)*self.__c_swx - (1-s)*t*self.__c_nwx \
        -s*(1-t)*self.__c_sex - t*s*self.__c_nex

    cy = - (1-s)*(1-t)*self.__c_swy - (1-s)*t*self.__c_nwy \
        -s*(1-t)*self.__c_sey - t*s*self.__c_ney

    x =  (1-s)*x_w + s*x_e + (1-t)*x_s + t*x_n  + cx
    y =  (1-s)*y_w + s*y_e + (1-t)*y_s + t*y_n  + cy
    return (x,y)
```

Listing 8.12: Class encapsulating the transfinite interpolation mapping for a square.

```
#!/usr/bin/env python
import sys
from GrAL.Cartesian2D.cartesian2d import Cartesian2D
from GrAL.Cartesian2D.mapped_geometry \
    import mapped_geometry_coord2 as mapped_geometry
from GrAL.View.View2DLibplot import View2DLibplot
```

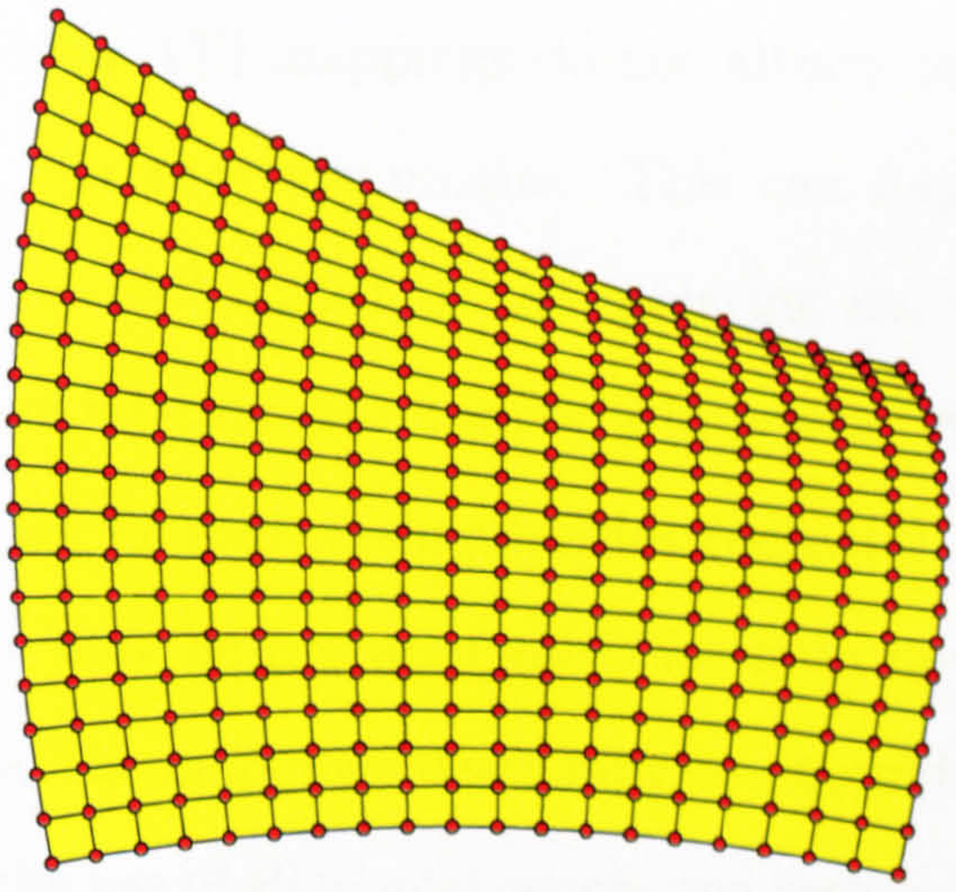


Figure 8.4: Grid defined by mapping shown in listing 8.13 using transfinite interpolation.

```
import math
from anulus import Anulus:
from tfi import BilinearBlendedSquare


grid = Cartesian2D(20,20)
anulus_mapper = Anulus()

square_mapper = BilinearBlendedSquare()
square_mapper.south = lambda x,y : (x, -0.2*x*(x-1))
square_mapper.north = lambda x,y : (x, 1+0.2*x*(x-1))
square_mapper.east = lambda x,y : (1-0.2*y*(y-1),-0.4*y*y+y)
square_mapper.west = lambda x,y : (0.2*y*(y-1),y)

# construct geometry for a grid
geom = mapped_geometry(grid, anulus_mapper.map)
# geom = mapped_geometry(grid, square_mapper.map)

viewer = View2DLibplot(type="X",params={"BITMAPSIZE":"300x300"})
viewer.setViewForGeometry(geom, border=10, radius=2)

# draw mesh
viewer.drawCells(geom)
viewer.drawVertices(geom)
```

Listing 8.13: Grid generation via algebraic or TFI mapping and subsequent grid visualisation.

It should be noted that TFI mappings do not always produce proper grids, as there is no guarantee that they are unique. This can happen for very distorted domains. Such cases can be handled by interpolating not only from the external boundaries but also from internal grid lines or grid points. Another problem is that TFI mappings propagate boundary singularities (corners) into an interior of the domain which property may be not acceptable for fluid flow solvers. This case is shown in Figure 8.5 and the mappings describing the boundaries are given in equation 8.3. In such cases the use of PDE grid generation techniques might be necessary.

north mapping:

$$x = s$$
$$y = 1$$

south mapping:

$$x = s$$
$$y = \begin{cases} 0 \text{ for } s < \frac{1}{3} \\ -3s + 1 \text{ for } \frac{1}{3} \geq s \leq \frac{2}{3} \\ -1 \text{ for } \frac{2}{3} < s \end{cases} \tag{8.3}$$

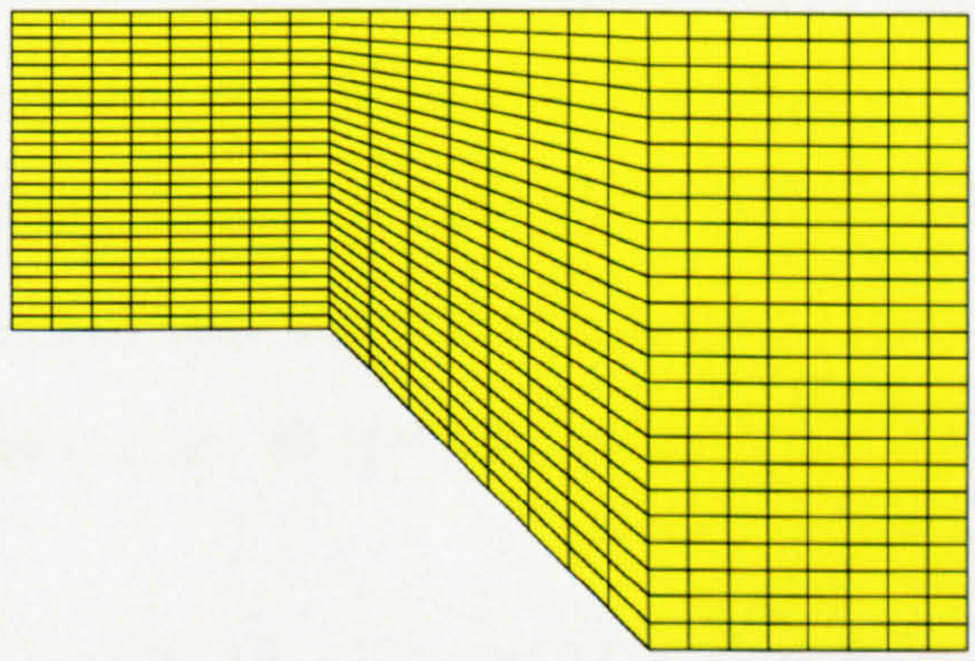


Figure 8.5: Propagation of boundary corners into a grid domain in the case of TFI grid generation.

east mapping:

$$x = 0$$

$$y = t$$

west mapping:

$$x = 3$$

$$y = 2t - 1$$

8.6 Mesh partitioning with PyGrAL

Several parallel algorithms for solving partial differential equations (PDE) are driven by geometric decomposition of a mesh over which the solution is sought. This means that the mesh is split into several regions, with each element assigned uniquely to a region. Such a decomposed mesh and the data defined on it are then usually distributed to several processors, and each processor concurrently applies the same processing to its share of the data. This parallel programming style is called SPMD (single program – multiple data) or the data partitioning approach as opposed to MPMD (multiple program – multiple data) or the task-parallel approach. Assuming the same performance characteristics of each processor, the effectiveness of SPMD schemes depend on equal distribution of processor load and minimisation of communication between processors. For many algorithms these conditions are equivalent to requiring an equal number of cells in each subdomain, and such a decomposition which ensures a minimum number of nodes on the subdomain boundaries. There are several mesh decomposition algorithms and several software packages for this task. The most popular one are METIS [98], Jostle [112], Chaco [137] and ParMetis [138].

While the GrAL library does not provide its own mesh decomposition implementation, it provides a generic interface to the METIS library, which hides the details of converting the generic GrAL mesh structures to the structure accepted by METIS.

```
from GrAL.Cartesian2D.cartesian2d import Cartesian2D
from GrAL.Cartesian2D.mapped_geometry \
    import mapped_geometry_coord2 as mapped_geometry
from GrAL.View.View2DLibplot import View2DLibplot
```

```
from GrAL.Cartesian2D.Partitioning import *
from GrAL.View.ViewPartitions import ViewPartitions
from anulus import Anulus
import math
import sys

if len(sys.argv) != 4:
    print "Usage %s nx ny npart" % sys.argv[0]
    sys.exit(1)

grid = Cartesian2D(int(sys.argv[1]),int(sys.argv[2]))

a = Anulus()
geom = mapped_geometry(grid, a.map)

viewer = View2DLibplot()
viewer.setViewForGeometry(geom, radius=2)

part = Partitioning(grid)
metis = MetisPartitioning()

nparts = int(sys.argv[3])

metis.calculate_cell_partitioning(grid, part, nparts)

colors = ['red','green','blue','yellow','magenta','brown','cyan']
ViewPartitions(viewer, part, colors)
```

Listing 8.14: Partitioning a grid using the GrAL interface to METIS library.

For each concrete data structure wrapped in PyGrAL a Python interface to GrAL partitioning module is provided. The use of a such module for the partitioning of a Cartesian2D grid is shown in listing 8.14. Besides calculating the partition data, PyGrAL also makes it easy to visualise them by providing the GrAL.View.ViewPartitions module. Figure 8.6 shows a partitioned mesh. It was created entirely with PyGrAL.

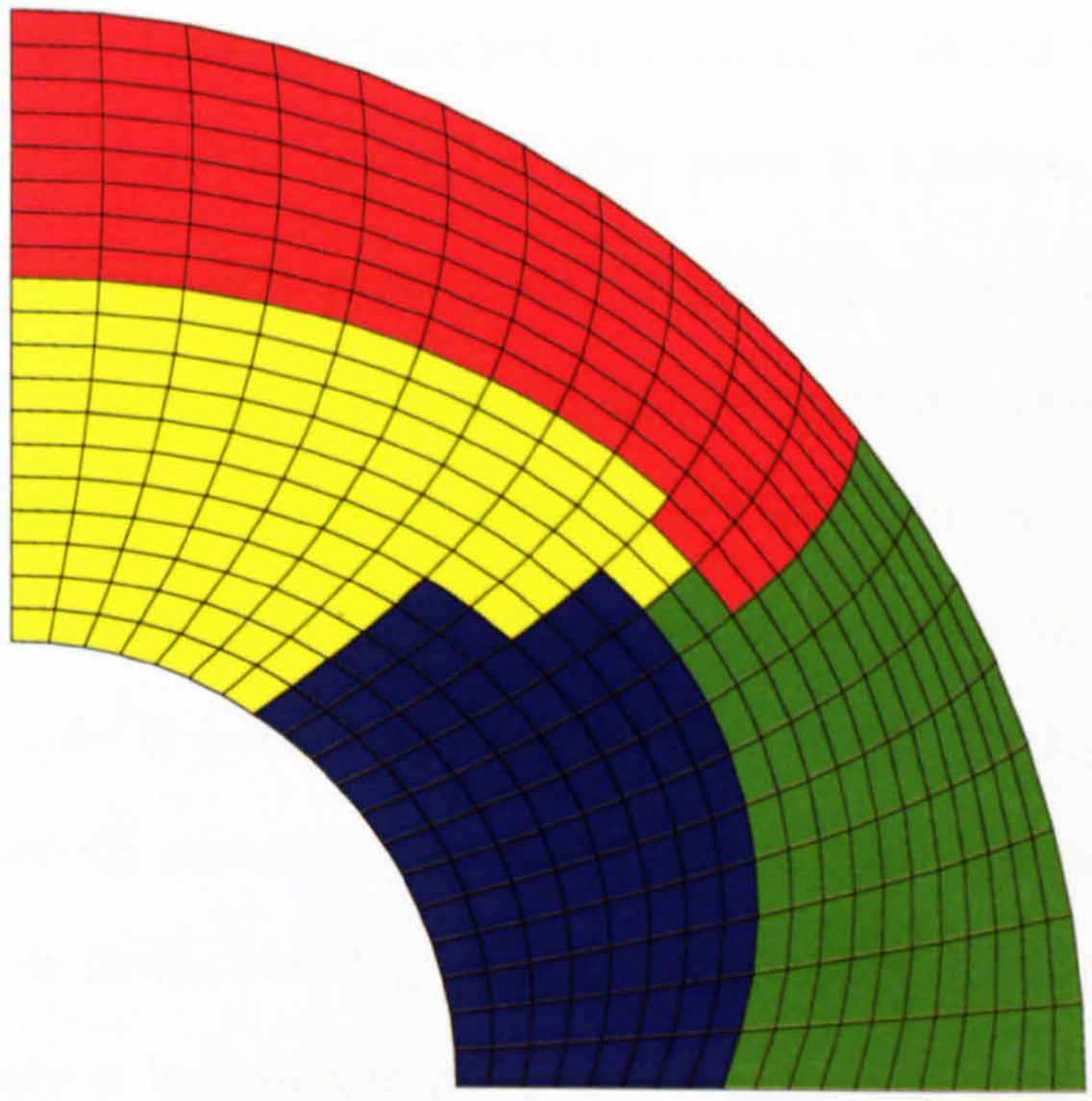


Figure 8.6: A partitioned grid produced by the program shown in listing 8.14.

8.7 Generation of block structured meshes with `cubic` and `PyCubic`

The software suite accompanying the book [5] contains very interesting block structured grid generator called `cubic`. It permits the generation of mixed triangular and quadrilateral meshes over geometric entities which are straight lines, quadrilateral cubic surfaces with curved sides, and triangular linear surfaces with straight sides. A unique feature of `cubic` is a versatile handling of line elements, because this generator was geared towards discretization of meshes for membrane structures with reinforcing cables.

Unfortunately `cubic` is distributed as a stand alone program which makes it difficult to use it as a GAGES component. Fortunately `cubic` is also an example of very clean design, thus it was not to hard to enclose the `cubic` functionality in a form of the C++ library called `cubic++`. The interface to this library is shown in appendix D.

In order to provide an even more user friendly interface `cubic++` was wrapped

in Python using SWIG. This interface permits `cubic` to be fed with data read from a file or to construct the `cubic` input piece by piece in a script.

Listing 8.15 shows the first possibility. In line 22 `cubic` is called to generate a mesh from the file given as a command line argument. The generated mesh is delivered in the `MESH` data structure from the E-Lib library. It is then saved in an output file with the help of the Python E-Lib wrapper as shown in lines 23, and 24. By itself this is hardly an improvement over the normal `cubic` usage, however the real benefit comes from the fact, that mesh generated by `cubic` can be interfaced with a `GrAL` based grid bus using the E-Lib adapter to the `GrAL` library. Consequently it becomes easy to create graphical back-end to `cubic` using the generic mesh viewing services provided by the `PyGrAL View` package. Lines 26–38 show the sample implementation of such back-end.

```
1  #!/usr/bin/env python
2
3  from optparse import OptionParser
4  from Cubic import CubicGenerator
5  from ELib import e_PutMeshData
6  from GrAL.Adapters.E_Lib.e_lib_adapter import *
7  from GrAL.View.View2DLibplot import View2DLibplot
8  import sys
9
10 usage = "usage: %prog [options] CUBIC_GEN_FILE"
11
12 parser = OptionParser(usage)
13 parser.add_option('-o','--output', type='string',dest='outfilename',
14                   metavar='FILE' , help="save output to FILE",
15                   default="/dev/stdout")
16
17 (options, args) = parser.parse_args()
18
19 if len(args) != 1:
20     parser.error("Incorrect number of agruments")
21
22 gen = CubicGenerator(args[0])
23 mesh = gen.GenerateMesh()
24 e_PutMeshData(mesh, options.outfilename)
25
```

```
26  emesh = MESHAdapter2D()
27  emesh.init(mesh)
28  if options.outfilename != '/dev/stdout':
29      figname = os.path.splitext(options.outfilename)[0] + ".eps"
30      figfile = open(figname, 'w')
31  else:
32      figfile = open('from_cubic.eps', 'w')
33
34  viewer = View2DLibplot(type="ps", outfile=figfile,
35                         params={"BITMAPSIZE":"500x500"})
36  viewer.setViewForGeometry(emesh, border=20, radius=4)
37
38  viewer.drawCells(emesh)
39  #viewer.drawVertices(emesh)
```

Listing 8.15: Simple Python front-end and back-end to the `cubic` mesh generator.

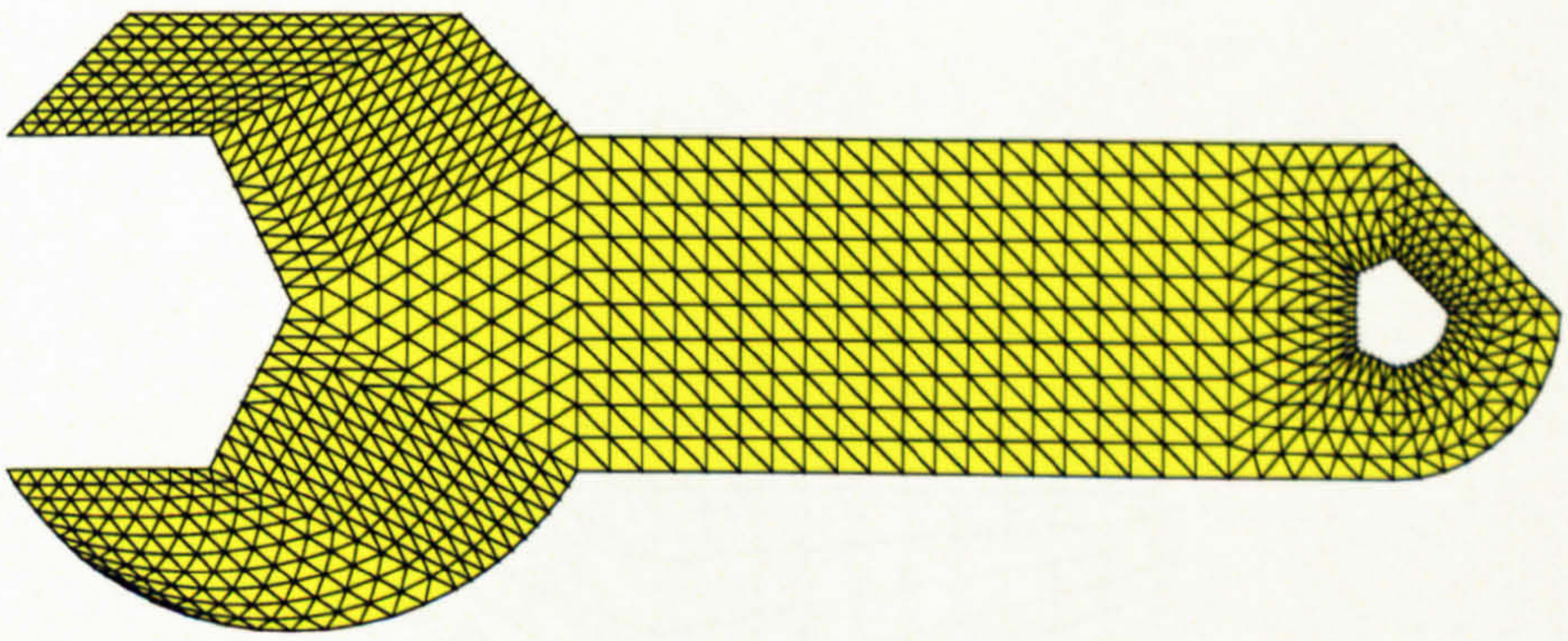


Figure 8.7: An example of a multiblock structured mesh produced by program from listing 8.15.

Another example of the gains of wrapping `cubic++` in Python is shown in listing 8.16. Here input to the `cubic` generator is constructed step by step. This increases the flexibility of `cubic`, making it easy to compute the geometric data of the blocks or to derive it from another geometric model. Additionally, listing 8.16 shows how mixed element meshes can be created with `cubic`.

```
from Cubic import CubicGenerator
from ELib import *
#from Eplx import eplx
gen = CubicGenerator(6, 2, 0)
nodes = [(0,0,0), (2,1,0), (2,2,0), (0, 3, 0), (-1, 1, 0), (-1,2,0)]
```

```
j=0;
for i in nodes:
    gen.SetNodeCoords(j, *i)
    j=j+1
gen.SetLinearPatch(0, [0,1,2,3], CubicGenerator.DEFAULT_ELEMENT)
gen.SetLinearPatch(1, [0,3,5,4])
gen.SetPatchElementType(1, CubicGenerator.TRIANG1)
gen.SetXDivision(0,10)
gen.SetYDivision(0,10)
gen.SetXDivision(1,10)
gen.SetYDivision(1,2)
mesh = gen.GenerateMesh()
e_PutMeshData(mesh, "ala")
```

Listing 8.16: An example of the direct manipulation of `cubic` input using a Python wrapper to `cubic++` library.

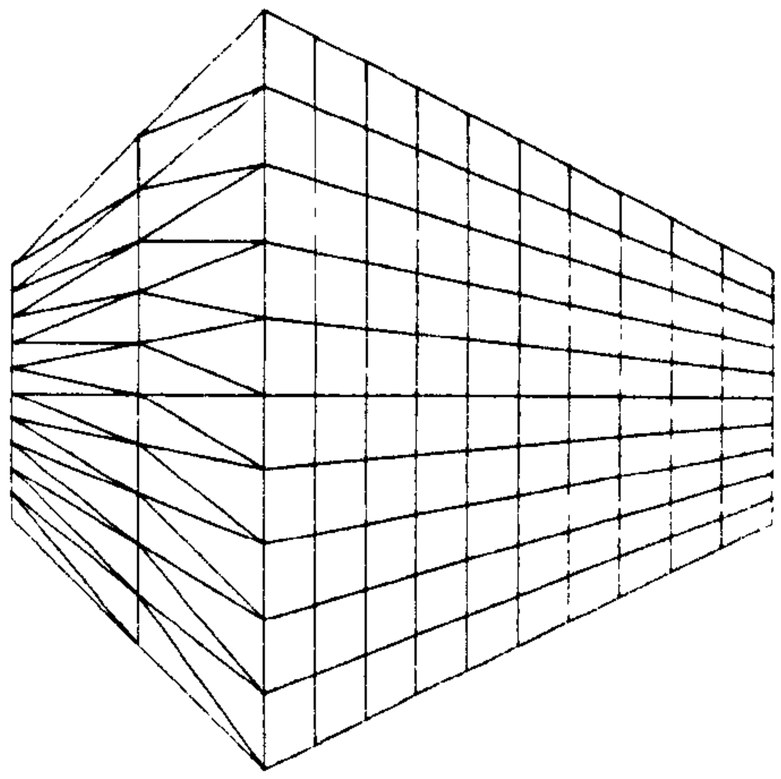


Figure 8.8: A block structured mesh with two kind of elements produced using the program from listing 8.16.

8.8 Handling triangulated surfaces with PyGTS

The GNU Triangulated Surfaces library [124] is a very interesting one, as it provides many advanced tools for dealing with triangulated surfaces, among them routines to design progressive meshes or to perform Boolean operations on triangulated surfaces.

The GTS GrAL adapter was not written as a part of this thesis but GTS can cooperate with other GAGES components thanks to its PyGTS Python interface.

Another advantage of providing the Python interface to GTS is that its interface becomes simpler. GTS uses the GLib object system [96] to support object oriented design and implementation. In `PyGTS` this object system is conveniently hidden behind Python classes.

The convenience of using the Python GTS interface is illustrated using two examples. The first one shows how to use the GTS implementation of incremental Dealunay triangulation and the `Libplot` based plotting solution to animate the triangulation algorithm for a set of vertices.

```
1  from GTS.Objects.triangles import *
2  from GTS.Objects.edges import *
3  from GTS.Objects.faces import *
4  from GTS.Objects.vertices import *
5  from GTS.Objects.surfaces import *
6  from GTS.Plot2D.plot2D import *
7  from GTS.Delaunay import *
8  import LPlot
9
10 def plot_edge(edge, plotter):
11     v1 = edge.segment.v1
12     v2 = edge.segment.v2
13     plotter.GetBasePlotter().fline(v1.p.x, v1.p.y, v2.p.x, v2.p.y)
14     return 1
15
16 def plot_vertex(vertex, plotter):
17     plotter.PlotPoint(vertex.p.x, vertex.p.y)
18     return 1
19
20 plotter = LPlot.LPlotter("ps")
21 gtsplotter = GtsPlotter2D(plotter)
22
23 coords = [[1,1,0],[3,3,0], [3,2,0],
24           [2,1,0], [3,1,0], [2,2,0],
25           [4,1,0], [1,4,0]]
26
27 vertices = []
28 for c in coords:
```

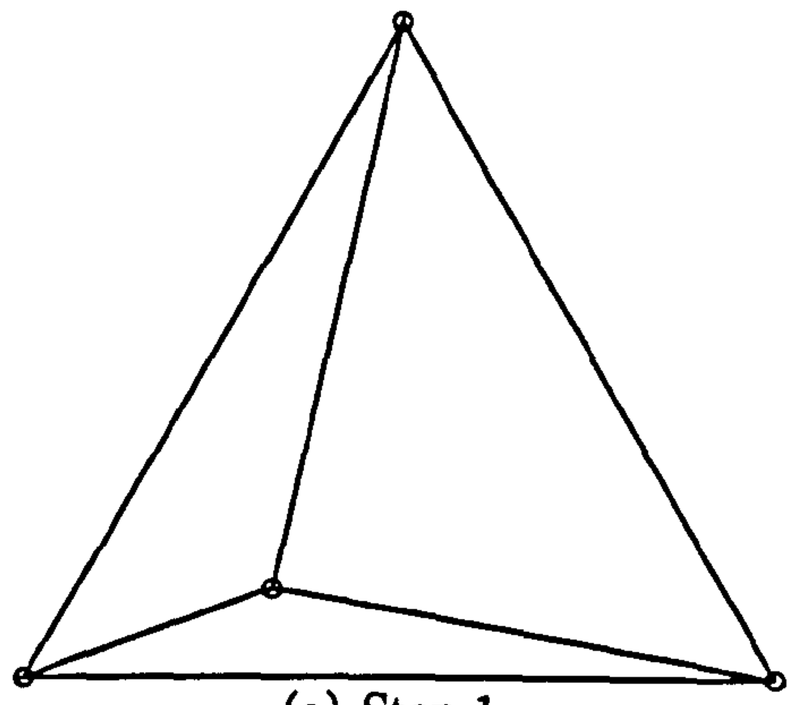
```
29      vertices.append(GtsVertex(*c))
30
31  triangle = gts_triangle_enclosing(gts_triangle_class(), vertices, 1.2)
32
33  surface = GtsSurface()
34  f = GtsFace(triangle.e1, triangle.e2, triangle.e3)
35  surface.add_face(f)
36
37
38  for v in vertices:
39      gtsplotter.Show(-2,-2,8,8)
40      gts_delaunay_add_vertex(surface, v, None)
41      gts_surface_foreach_edge(surface, plot_edge, plotter)
42      gts_surface_foreach_vertex(surface, plot_vertex, plotter)
43      gtsplotter.BasePlotter().closepl()
```

Listing 8.17: Animating the incremental Delaunay triangulation algorithm using PyGTS interface to GTS library.

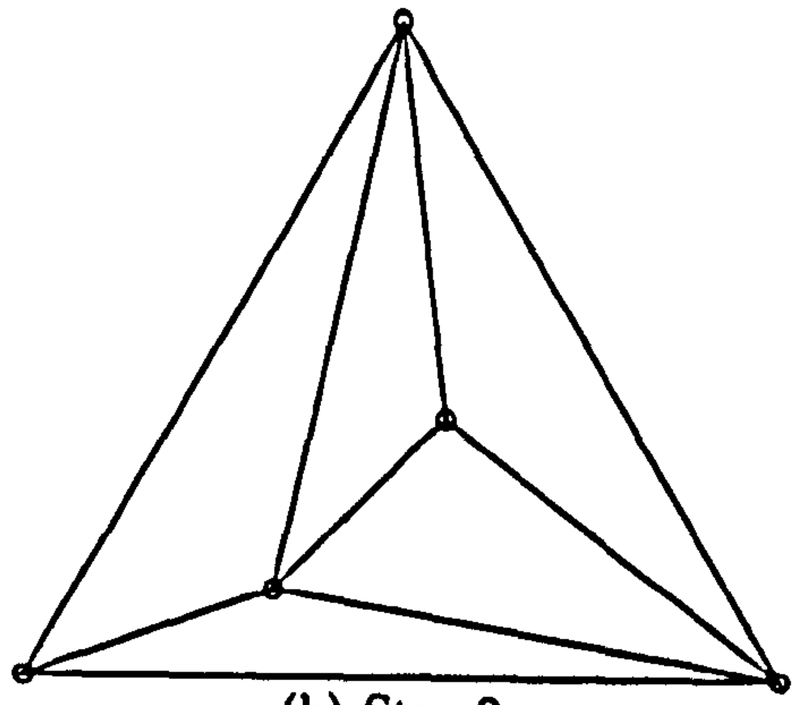
In listing 8.17 lines 23–25 define a set of vertex coordinates. Lines 27–29 create a list of `GtsVertex` objects. That list is then used to calculate a triangle enclosing all vertices, line 31. This triangle is used as the initial triangle of the triangulation, in lines 33–35. Finally, in lines 38–43, the remaining vertices are added one by one to the triangulation, plotting, after each added vertex, the current triangulation configuration. Figure 8.9 shows six initial steps of the triangulation process.

The second example shows how easily one can build an OpenGL viewer for triangulated surfaces stored in GTS file format. GTS itself does not provide any rendering capabilities but it is quite simple to program them. Appendix E contains the complete source code for the `GtsGLRenderer` class translating the GTS surface into an OpenGL display list. For convenience it was implemented in Python, but it can be easily translated to C, if the performance is not satisfactory (which hardly should be a case because of the use of the OpenGL display list).

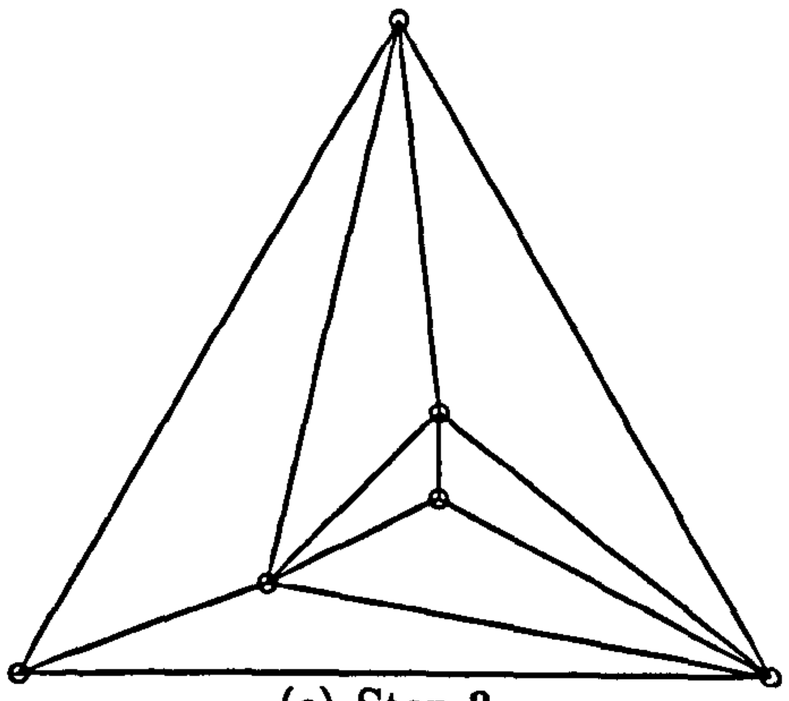
```
#!/usr/bin/env python

from OpenGL.GL import *
from OpenGL.Tk import *
```



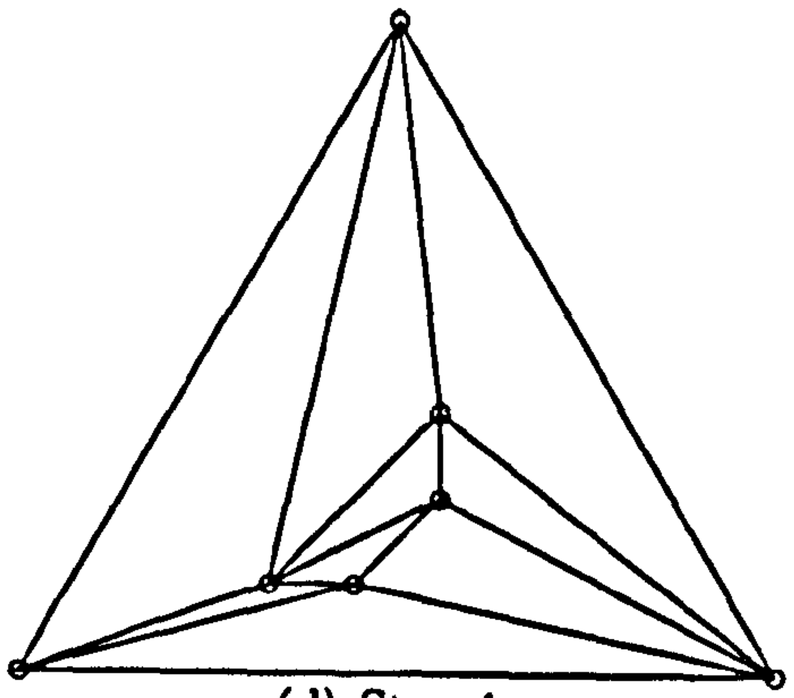
(a) Step 1



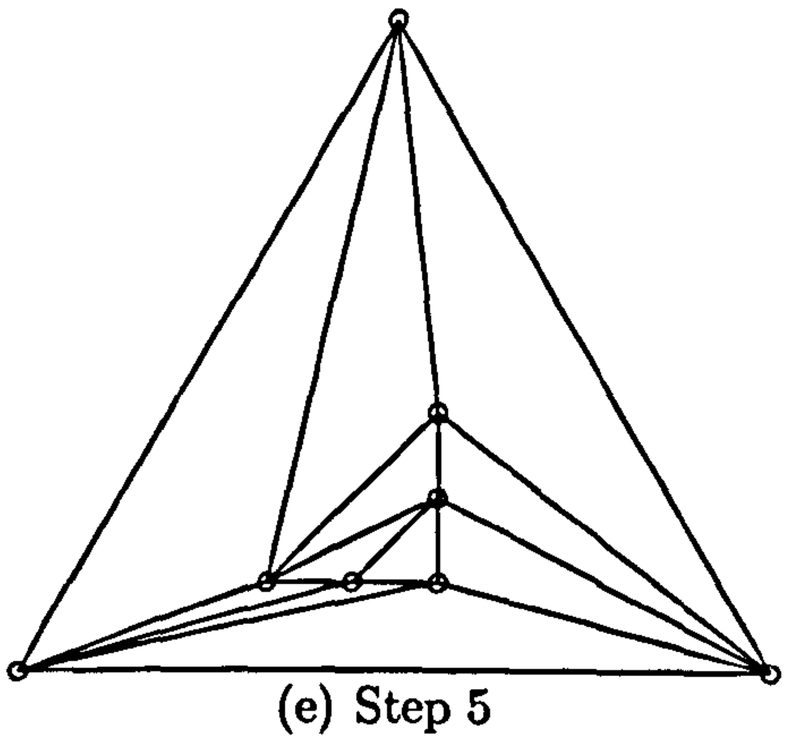
(b) Step 2



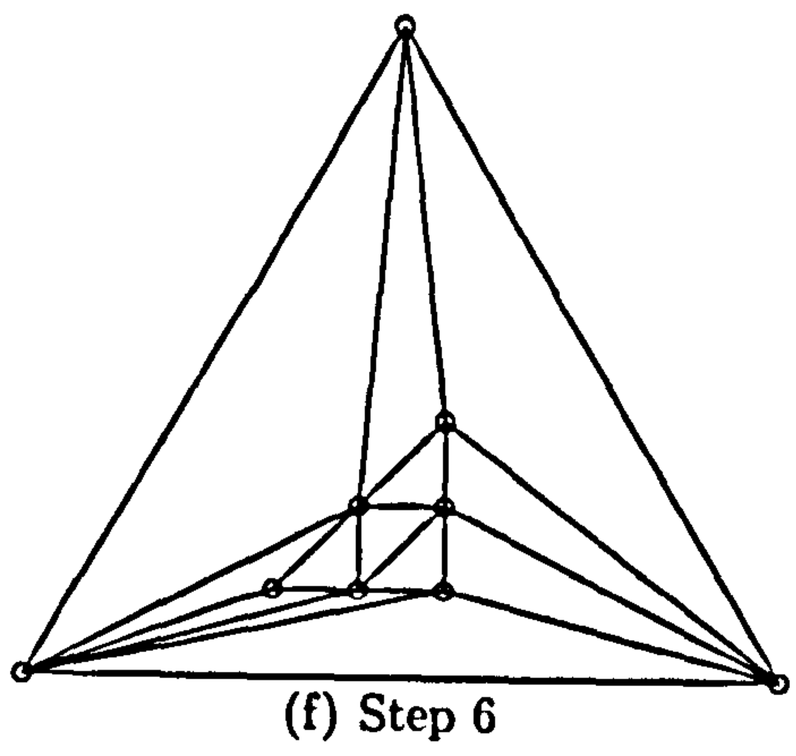
(c) Step 3



(d) Step 4



(e) Step 5



(f) Step 6

Figure 8.9: Illustration of the incremental Delaunay triangulation produced by program shown in listing 8.17.

```
from GtsGLRenderer import *
from GTS.Objects.vertices import *
from GTS.Objects.edges import *
from GTS.Objects.faces import *
from GTS.Objects.surfaces import *
from GTS.Misc.files import *
import sys
import Tkinter
```

```
class GtsViewer(Tkinter.Frame):
    def Display(self, event=None):
        self.__setup_gl()
        self.mapper.Render()

    def __setup_gl(self):
        glClearColor(0.1, 0.5, 0.5, 0)
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glOrtho(0,1,0,1,0,1)
        glLightfv(GL_LIGHT0, GL_AMBIENT, (100, 100, 100, 0))
        glShadeModel(GL_FLAT)

    def __setup_gl_widget(self):
        self.gl = Opengl(self,width = 500, height = 500, double = 1, depth = 1)
        self.gl.pack(side = 'top', expand = 1, fill = 'both')
        self.gl.set_centerpoint(0, 0, 0)
        self.gl.set_eyepoint(100)
        self.gl.redraw = self.Display

    def __init__(self, mapper, **kw):
        Tkinter.Frame.__init__(self)
        self.mapper = mapper
        self.__setup_gl()
        self.__setup_gl_widget()
        #self.gl.set_background(0.2, 0.5,0.5)

        self.pack()
        self.gl.tkRedraw()
        self.gl.mainloop()

def main():
    file = open(sys.argv[1], "r")
    s = GtsSurface()
    gf = GtsFile(file)
    gts_surface_read(s, gf)
    mapper = GtsGLRenderer(s)
    ogl = GtsViewer(mapper)

main()
```

Listing 8.18: Example of using PyGTS and GtsGLRenderer for building the GTS file format viewer.

Listing 8.18 provides the complete code for the viewer. The GUI part comes from the built-in Tkinter module. Handling of OpenGL window, OpenGL initialisation and user interaction is supported by modules from the PyOpenGL package. Reading the data file and creating the appropriate surface data structure is done by GTS. Actually the user has to do nothing else but to put all the bricks together. This is a perfect example of the benefits gained from having a set of compatible components.

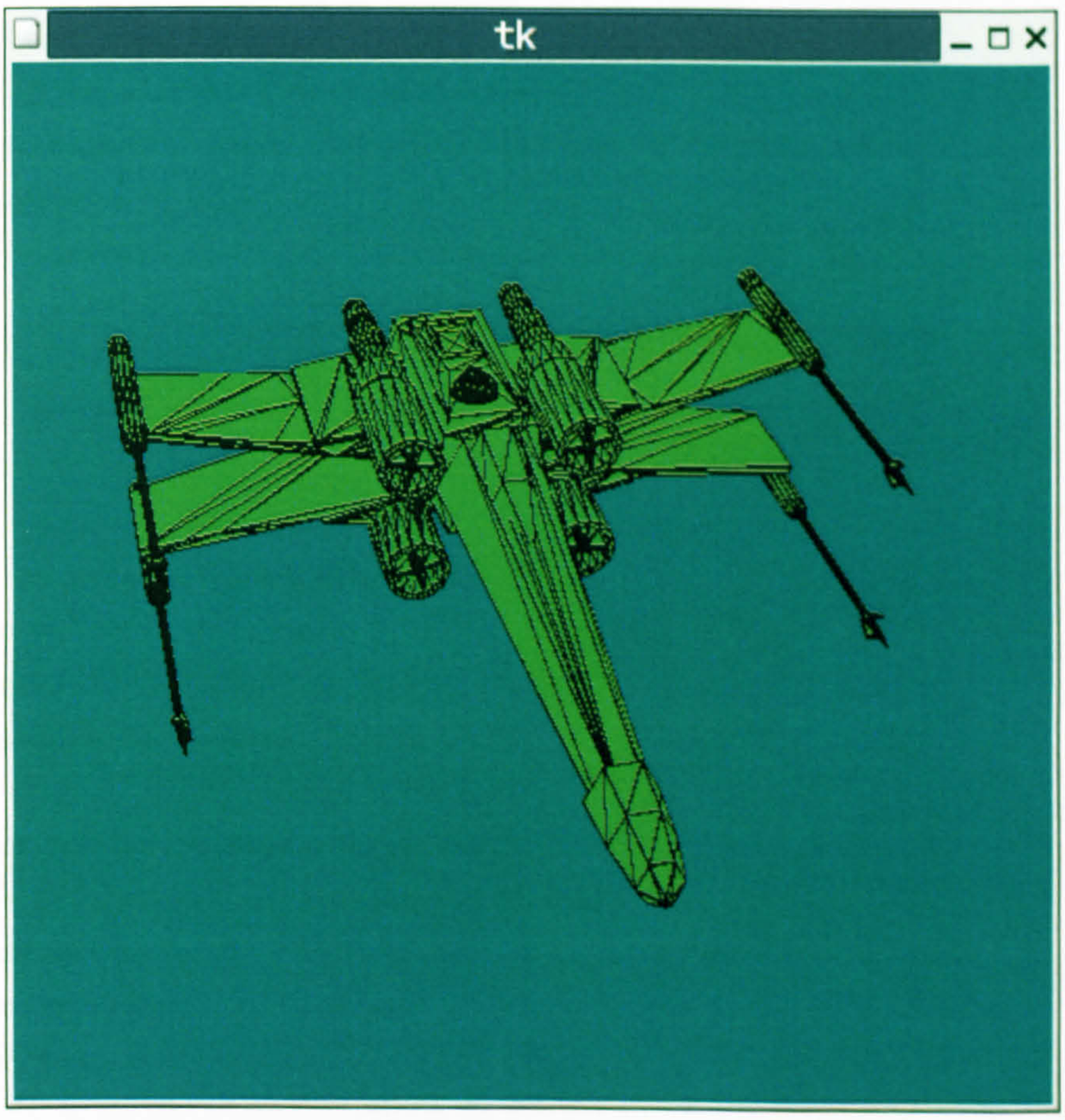


Figure 8.10: OpenGL rendering of triangular surface model of the Star Wars' X-fighter produced using the program shown in listing 8.18.

8.9 Python interface to OpenNURBS

At the time of writing this thesis the OpenNURBS library was distributed only with a C++ interface. Being chosen as the basis for the geometry bus it is necessary to

provide a scripting interface to it. Fortunately a very clean interface makes it possible to build a Python wrapper for it using SWIG. In this way, a very powerful tool for defining curved objects and for experimenting with their geometric transformations was obtained. The first example illustrating this claim is shown in listing 8.19. In this listing a Bezier curve is created, and its rotated and translated copies are plotted using functions from the `ONPlot` utility library discussed in section 8.3.

```
from ONPlot2D.on_plot import *
from  ONPlot2D.on_plotter import ONPlotter
import PyON.circle
import PyON.point
import PyON.on_array
import PyON.bezier
import math

PyON.Start()

ctrl = PyON.on_array.ON_2dPointArray()
points = [(0.0, 0.0), (1.0, 0.0), (1.0,1.0), (0.0, 1.0)]
for p in points:
    ctrl.AppendNew().Set(*p)

bezier = PyON.bezier.ON_BezierCurve(ctrl)
bezier_bis = PyON.bezier.ON_BezierCurve(bezier)
plotter = ONPlotter("ps")
plotter.Show(-0.1, -0.1, 1.1, 1.1)
pt = plotter.GetBasePlotter()

pt.pencolorname("red")
PlotBezierControlPolygon(pt,bezier)
pt.pencolorname("blue")
PlotBezierUniform(pt,bezier,20)

center = PyON.point.ON_3dPoint(0.5, 0.5, 0);
axis = PyON.point.ON_3dVector(0,0,1);
bezier.Rotate(math.radians(90), axis, center)

pt.pencolorname("magenta")
PlotBezierUniform(pt,bezier,20)

v = PyON.point.ON_3dVector(0.1, 0,0);
```

```
bezier_bis.Translate(v)
pt.pencolorname("green")
PlotBezierUniform(pt,bezier_bis,20)
PyON.Finish()
```

Listing 8.19: Creating and transforming Bezier curve.

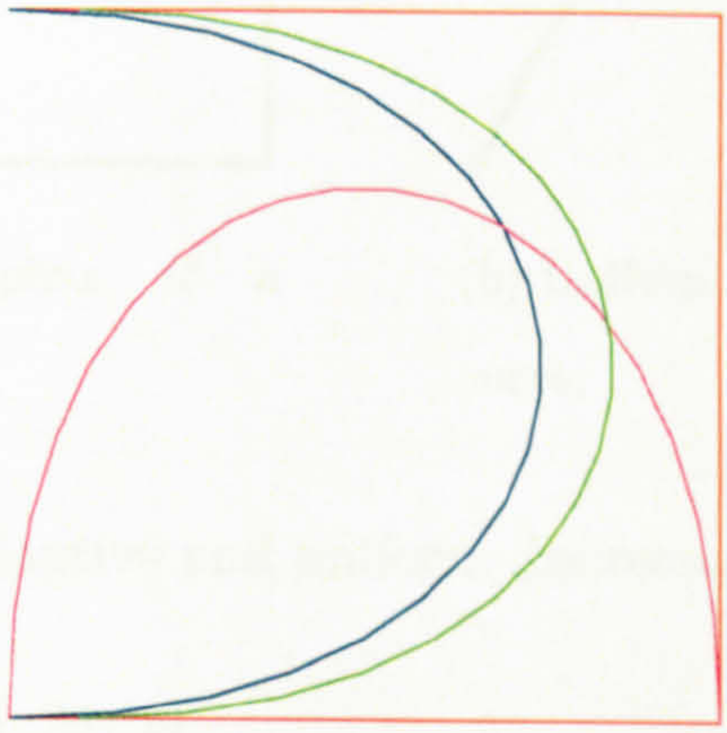


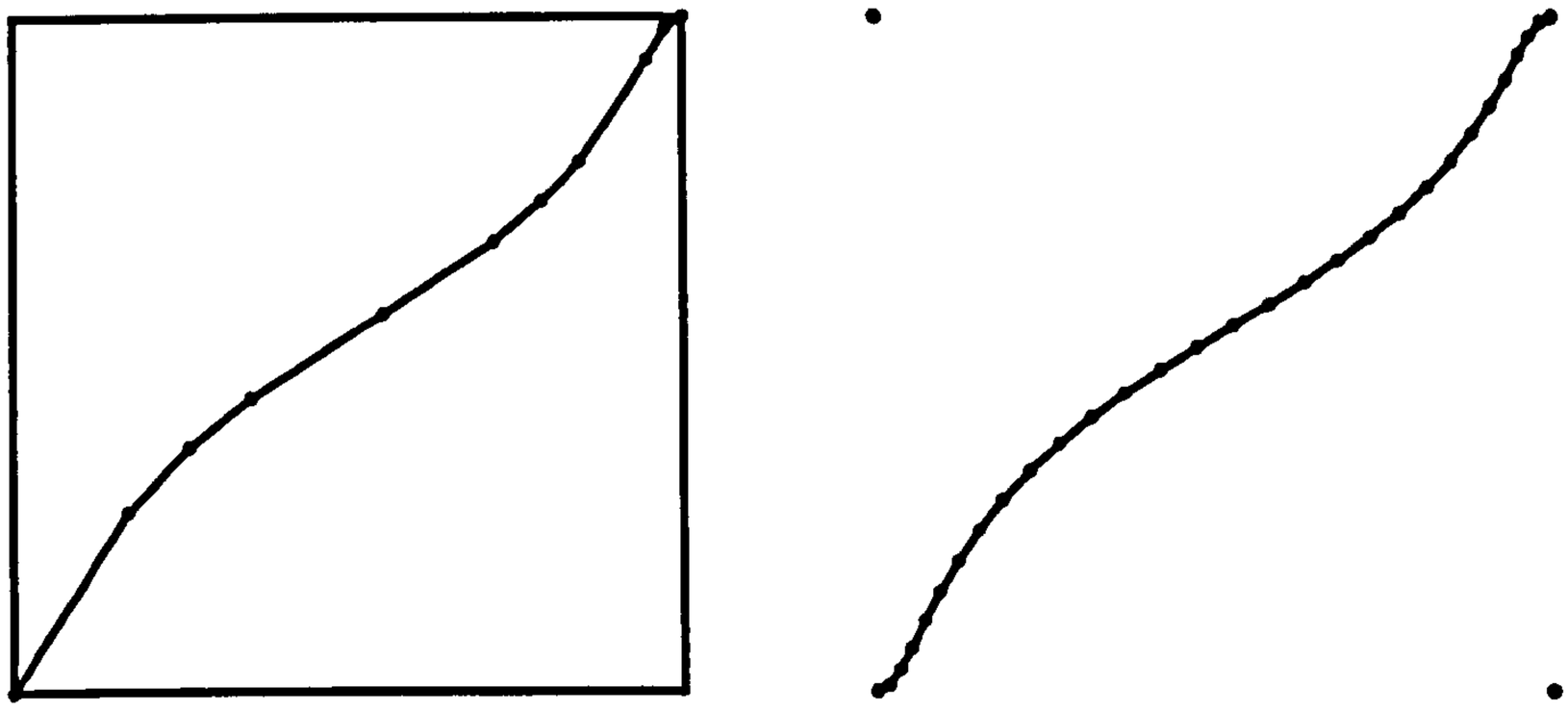
Figure 8.11: Results of the transformations of the Bezier curve from the program shown in listing 8.19: blue – original curve, green – translated curve , magenta – rotated curve. Curve control polygon is plotted in red.

Another example of how Python interface is so useful to OpenNURBS, is shown in figure 8.9. It illustrates the the properties of adaptive and uniform curve sampling algorithm. Other examples of using Python bindings to OpenNURBS are shown in the next Chapter, where the Python functions and classes are used to control the mesh generation process.

8.10 Other Python interfaces

Besides the interfaces described above the scripting layer of GAGES contain a few more. One of the principles of GAGES, though not always easy to fulfil, is that everything that is possible in the C/C++ layer should be possible in the scripting language layer. Based on this, the following Python interfaces were created:

- Direct interface to E-Lib's `MESH` data structure. Examples of using it are given



(a) Adaptive sampling of a NURBS curve.

(b) Uniform sampling of the same curve.

Figure 8.12: Example of adaptive and uniform discretization of a parametric curve.

in listing 8.15, lines 6, 23, 24.

- Interface to E-Lib's GrAL adapter. It provides another way of manipulating MESH data structure. An example of using it is shown in listing 8.15, lines 26, 27, 36, and 38.

- Interface to OpenDX output filter. This is interface to the filter is discussed in section 7.6.3 and illustrated in listing 7.6.

- Interface to the unstructured grid adapter for VTK. This is interface to the filter discussed in section 7.6.3 and illustrated in listing 7.5 as well as used to create figures 9.8, 9.9 and 9.10

- Interfaces to `triangle` and `GRUMMP` `GrAL`'s adapters described in sections 7.6.1 and 7.6.2, respectively.

8.11 Summary

This Chapter provided the description of how to make GAGES a hybrid system, that is, how to build a scripting language interface to GAGES components. Building a hybrid system requires selecting a scripting language and interface generation tool.

In this Chapter Python and SWIG were selected, respectively. Examples of building Python interface to various complex features of the GrAL library, such as iterators or template classes are the proof, that selecting SWIG as the interface generation tool was the right decision.

The effective wrapping of a such large library as GrAL required to introduce some SWIG macros to factor out common task. Some of these macros are shown in section 8.2.2 and 8.2.3.

Selecting Python as the extension scripting language turned to be the right decision too. This was illustrated by the ease and flexibility of building TFI based mesh generator presented in section 8.5.

This Chapter presented also the examples of building the Python interfaces to the other GAGES components such as a plotting library, a surface triangulation library, a structured mesh generator and visualisation programs. These examples form the evidence that it is possible to achieve one of the main goals of GAGES, that is, to use it as a rapid prototyping and tools building environment. Besides the examples concerning grid based components, this Chapter described also Python interface to the OpenNURBS library which was selected as the underlying library for the geometry bus.

The techniques and tools developed for this Chapter will be used in the next Chapter to enable and support building a surface mesh generator. So far the grid bus and the geometry bus were treated as separated. However the main strength of GAGES lies in connecting them. This connection is also the main difficulty to overcome. The next Chapter presents the development of the software tools which will make such connection possible.

Chapter 9

Surface mesh generation

Besides linking various components for geometry or grid manipulations, the strength of the GAGES environment lies in the connection between the geometry and grid buses. As indicated in Chapter 5 the effective FEM analysis of three dimensional cases requires the problem to be defined on a geometric model and a mesh to be generated in an automatic way. Surface mesh generation comprises and important part in linking geometric model and its grid discretization. The surface mesh generator is thus an indispensable component for addressing three dimensional problems in computational mechanics.

This Chapter presents development of a surface mesh generator. However, the main goal is not improvement of an existing algorithm or designing a new one, but establishing how the existing GAGES components can be used to provide an inexpensive, simple and workable solution. Though superficially it may seem as an unrelated research topic, the investigation reported in this Chapter is in fact a crucial part of the study on the feasibility of the GAGES architecture. It is crucial in the sense that it shows what is necessary to implement a link between two main parts of GAGES: geometry bus and grid bus. The place of this Chapter in the whole discussion on GAGES is shown in Figure 9.1.

The development of a surface mesh generator will be also a good test case for showing the usefulness of the tools developed in Chapters 6, 7 and 8.

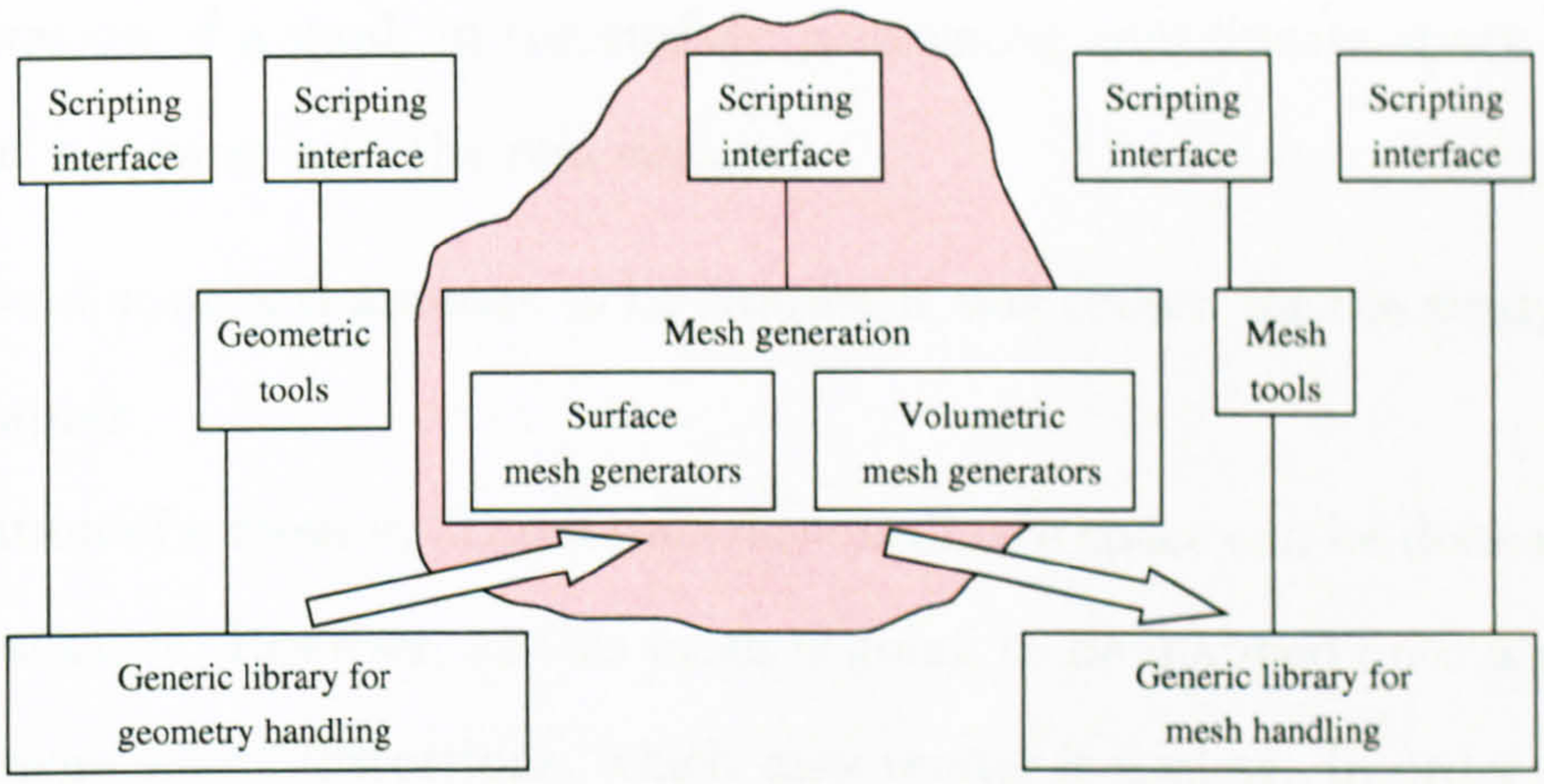


Figure 9.1: The place of this chapter in the whole discussion on GAGES.

9.1 Linking geometry and grid buses

Mesh generators are undoubtedly the most important components of GAGES, as they link geometry and gird buses. Research on meshing algorithms and building automatic generators has been the subject of intensive study for a long time. This resulted in many software packages for 2D and 3D meshing. However, open source surface meshing packages are quite rare, as they are the most difficult ones to write, because of nonlinear surface geometry that has to be handled. At the same time surface meshing is a necessary prerequisite for most of three-dimensional meshing algorithms.

It is commonly realised that mesh generators provide mapping from geometric elements to grid elements. However, for many applications, it is equally important to provide mapping in the opposite direction, that is to be able to identify geometric object from which a given grid element was derived. This is important if one has to update geometry or has to program adaptive mesh refinement. Good mesh generators usually provide tagging systems which allow data to propagate from the geometric model onto a discrete model.

There are two main approaches to the generation of surface meshes:

- generation of a mesh directly on the surface,

- generation of a mesh in the surface parametric coordinates space and subsequent mapping onto the real surface.

As the second approach appears to be simpler it was chosen for the study presented in this Chapter.

Generation of a mesh in 2D parametric coordinate space can be done using many different packages. However, as this mesh is going to be mapped onto a real surface in can undergo severe distortions, which may render it useless. In order to avoid it, mesh in a parametric space has to have special properties. In order to capture the geometric features of the surface it has to be dense in regions of high curvature, and in order to compensate mapping distortion, elements in a parametric space must be counter-distorted so after they are mapped on the surface, regular element shape is restored. Thus the parametric space mesh generator has to provide the following features:

- flexible pointwise control of the mesh density,
- ability to generate anisotropic meshes governed by variable metrics.

None of the popular open source meshing programs known to the author provided the required features, so it was decided that the `Triangle` [106] mesh generator will be modified to add them.

9.2 Nonuniform mesh generator

`Triangle` is a very versatile 2D Delaunay mesh generator. The mesh density can be controlled either globally by setting the maximum allowable element area or locally by assigning an area constraint to polygonal regions. Neither of these ways suffice for surface mesh generation. However, the `Triangle` generator can be used as a library, and in this case it provides a hook for the user's callback function which decides when the mesh refinement process is completed. This callback function has the following signature:

```
typedef REAL *vertex;
int triunsuitable(triorg, tridest, triapex, area, userData)
              vertex triorg;    /* The triangle's origin vertex. */
              vertex tridest;   /* The triangle's destination vertex. */
              vertex triapex;   /* The triangle's apex vertex. */
              REAL area;        /* The area of the triangle. */
```

On output this function should return 0 if the element size is acceptable or 1 if it needs further refinement.

In order to use the above feature the `TriangleGenerator` class discussed in section 7.6.1 was modified, and turned into a template class parameterised by a class called `TriangleAccepter`. The new interface is shown in listing 9.1.

```
1  template <class TriangleAccepter>
2  class TriangleGeneratorPro : public TriangleGenerator {
3     public :
4     TriangleGeneratorPro() {}
5     TriangleGeneratorPro(TriangleAccepter const& accepter)
6       : accepter_(accepter) {}
7
8     void SetTriangleAccepter(TriangleAccepter const& accepter);
9     TriangleAccepter const & GetTriangleAccepter() const;
10    int triunsuitable(const double *pa, const double *pb, const
11                                  double *pc, const double area) const {
12       return accepter_(pa, pb, pc, area);
13    }
14    private:
15       TriangleAccepter accepter_;
16 };
```

Listing 9.1: Interface to the modified `TriangleGenerator` endowed with refined mesh size control mechanism.

The new interface requires that a class to be used as the `TriangleAccepter` have to provide the method of signature shown in listing 9.2, which will be used to govern the triangulation process. This way one can then easily write a custom function object to control the mesh density.

```
int operator()(const double *pa, const double *pb, const
```

```
double *pc, const double area);
```

Listing 9.2: Signature of mesh density control function

To make the use of the new generator even simpler a Python interface to it has been provided. Hence `TriangleAccepter` can be written as a Python class and easily changed at run time. Listing 9.3 shows how flexibly one can control the mesh density from the Python script.

```
1  class MeshDensityControler:
2      def __call__(self, a, b, c, area):
3            s = map(lambda x,y,z: (x+y+z)/3.0, a,b,c)
4            radius = sqrt(s[0]*s[0] + s[1]*s[1])
5            period = 2*pi
6            if area < 0.0005 + 0.003*fabs(cos(period*radius)):
7               return 0
8            else:
9               return 1
10 controller = MeshDensityControler()
11
12 generator = TriangleGeneratorPro()
13 generator.SetTriangleAccepter(controller)
14 adapter = TriangleAdapter()
15
16 flags = 'a0.1q20cpzjVu'
17
18 generator.SetOptions(flags)
19 generator.Triangulate(input, flags, adapter)
```

Listing 9.3: Flexible mesh density control from the Python script.

Class `MeshDensityControler` defines the mapping:

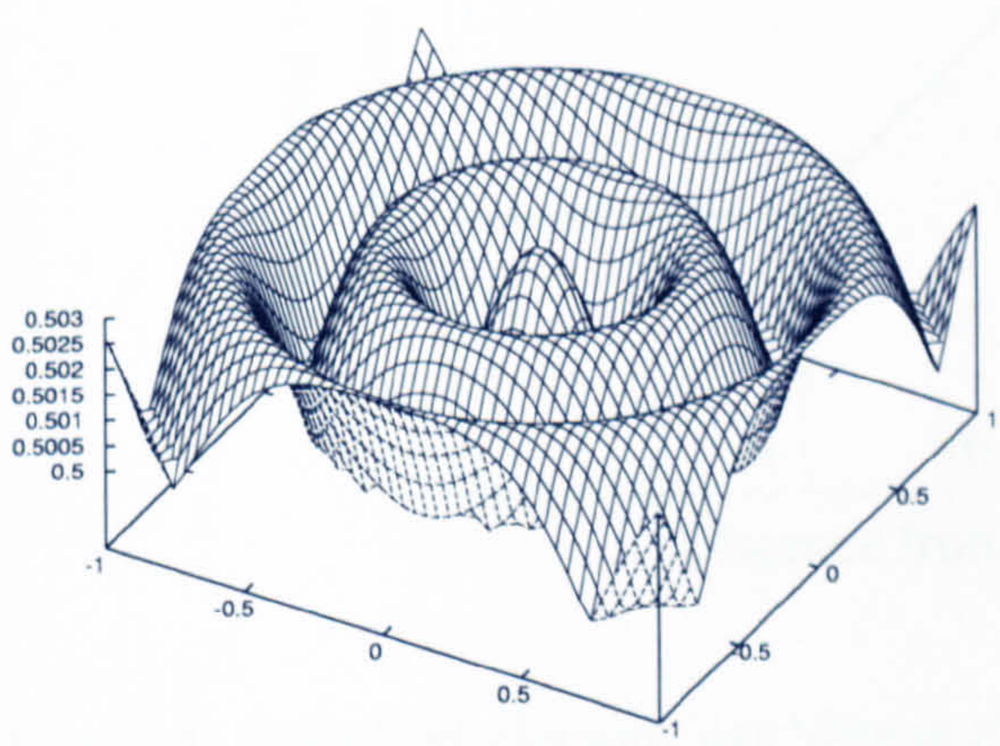
$$f : (x, y) \rightarrow \text{required triangle area} \in R$$

$$f : (x, y) = 0.0005 + 0.003 * |\cos 2\pi r| \tag{9.1}$$

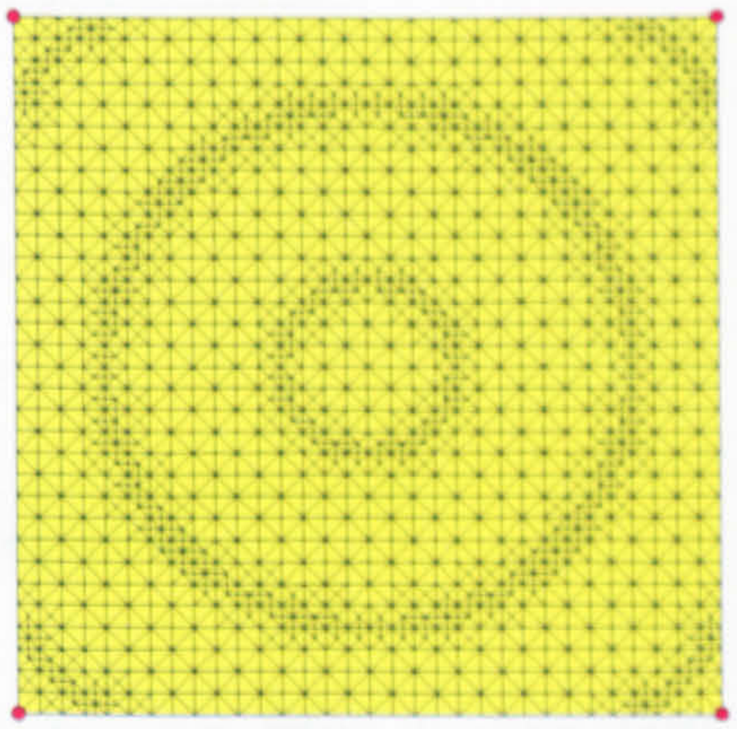
$$r = \sqrt{x^2 + y^2}$$

If for a triangle T $Area_T < f(x_T, y_T)$ where (x_T, y_T) is a triangle gravity centre then such triangle is accepted. Figure 9.2a) shows the size constraint function defined in listing 9.3 and the grid over a square region generated according to that mapping.

More sophisticated forms of mesh density control are possible. Figures 9.2a) and



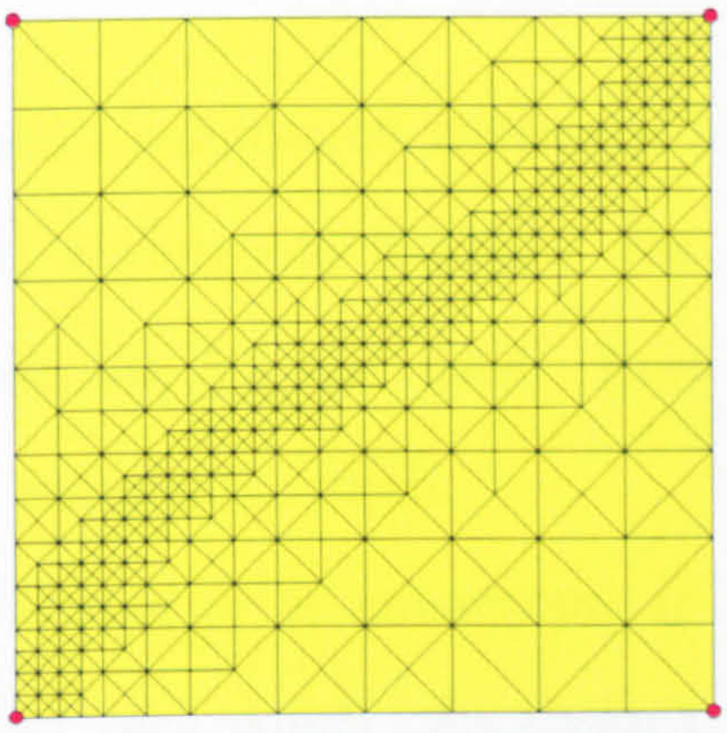
(a) Triangle area control function



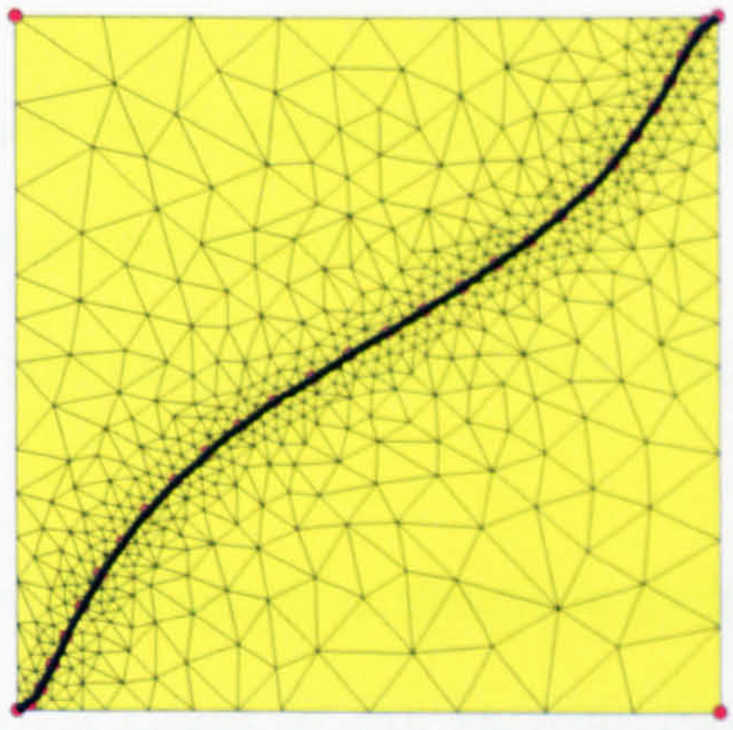
(b) Generated mesh

Figure 9.2: Triangulation with triangle area controlled by function given in equation 9.1

9.2b) show the mesh in a square domain refined around a NURBS curve. In this



(a) Mesh generated without curve points.



(b) Mesh generated with curve points.

Figure 9.3: Mesh refined around a NURBS curve.

case mesh density is a function of the distance of the triangle centre to the curve and takes the form illustrated in Figure 9.4. Additionally, in case presented the distances d_1 and d_2 were made dependent on the curve resolution. Figure 9.2 shows two cases – one when the curve points are not included in the triangulation and the

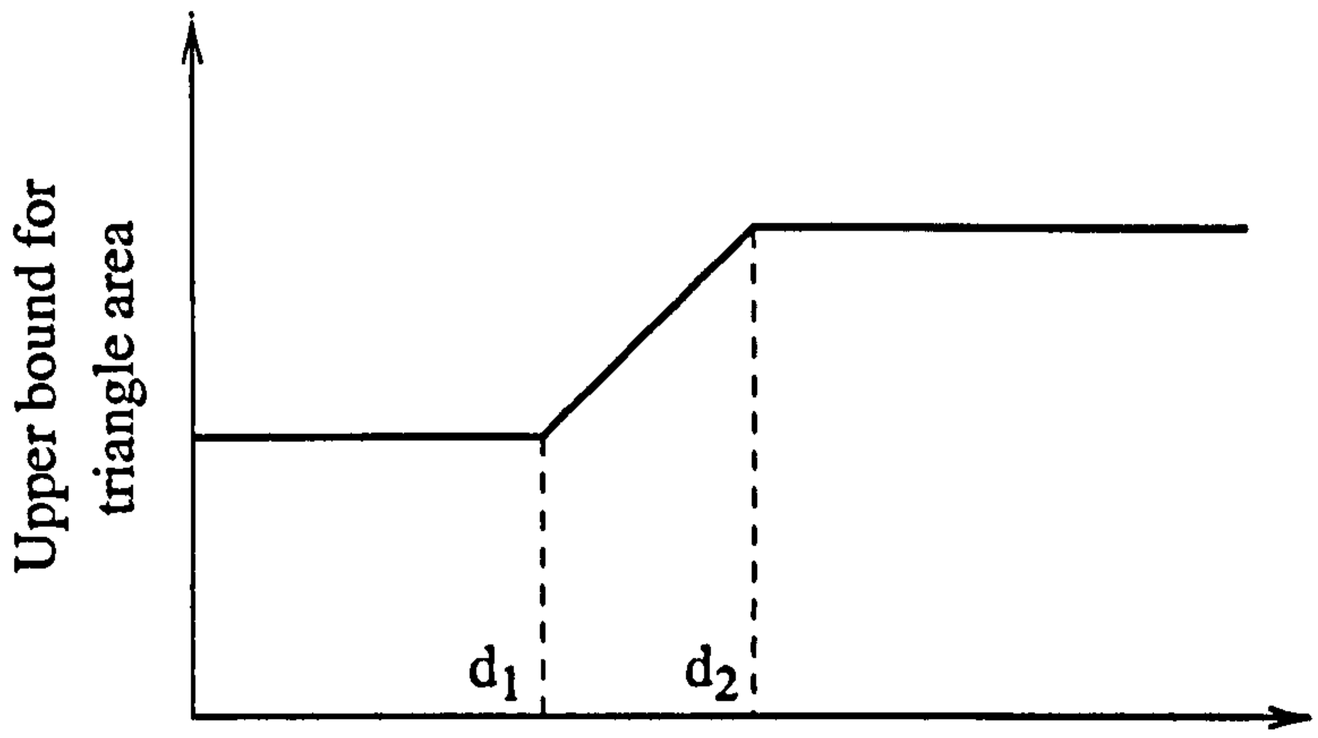


Figure 9.4: Required element are versus element distance from guiding curve.

other, when they are added to the triangulation. All computations involving the NURBS curve were done using the OpenNURBS library and its Python wrapper.

9.3 Anisotropic mesh generator

The second modification necessary for surface mesh generation was enabling generation of anisotropic meshes. For that, the framework presented in references [31, 32, 62, 49, 50] was used. Here we introduce an informal but intuitive definition of anisotropic mesh:

A mesh is said to be anisotropic if one can distinguish a direction (which can be different at each point) along which the longest edges of the surrounding elements are aligned. If the orientation of the longest edges is random, then the mesh is called isotropic. This concept is illustrated on figures 9.5a) and 9.5b).

Figure 9.5 shows a fixed set of points – the convex hull of these points will be triangulated (without adding new points) taking also a square hole into account. Figure 9.6a) shows an isotropic mesh. Here orientation of the triangles is random, though symmetric due to the symmetry of input points. Figure 9.6b) shows an anisotropic mesh defined on the same set of points. One can easily distinguish one direction along which triangles are elongated.

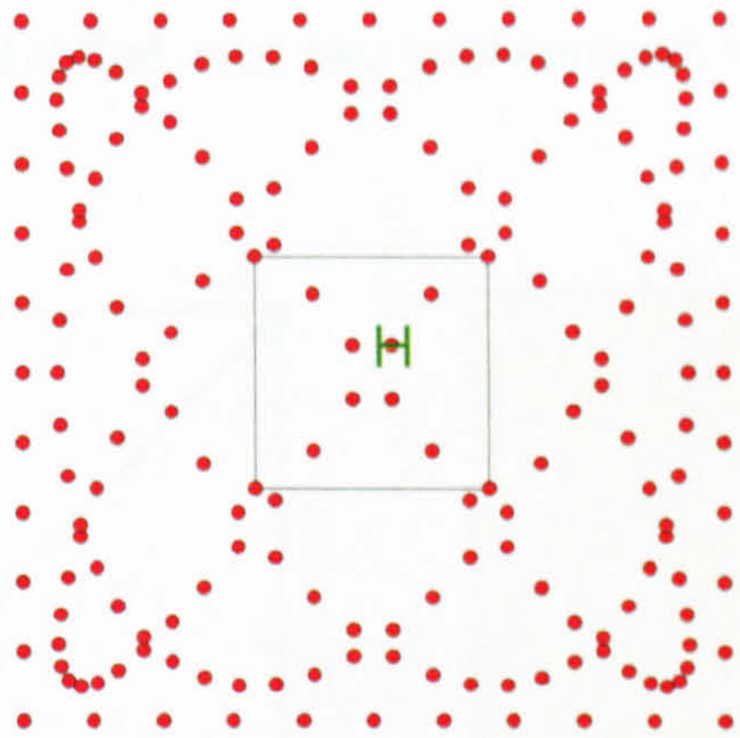
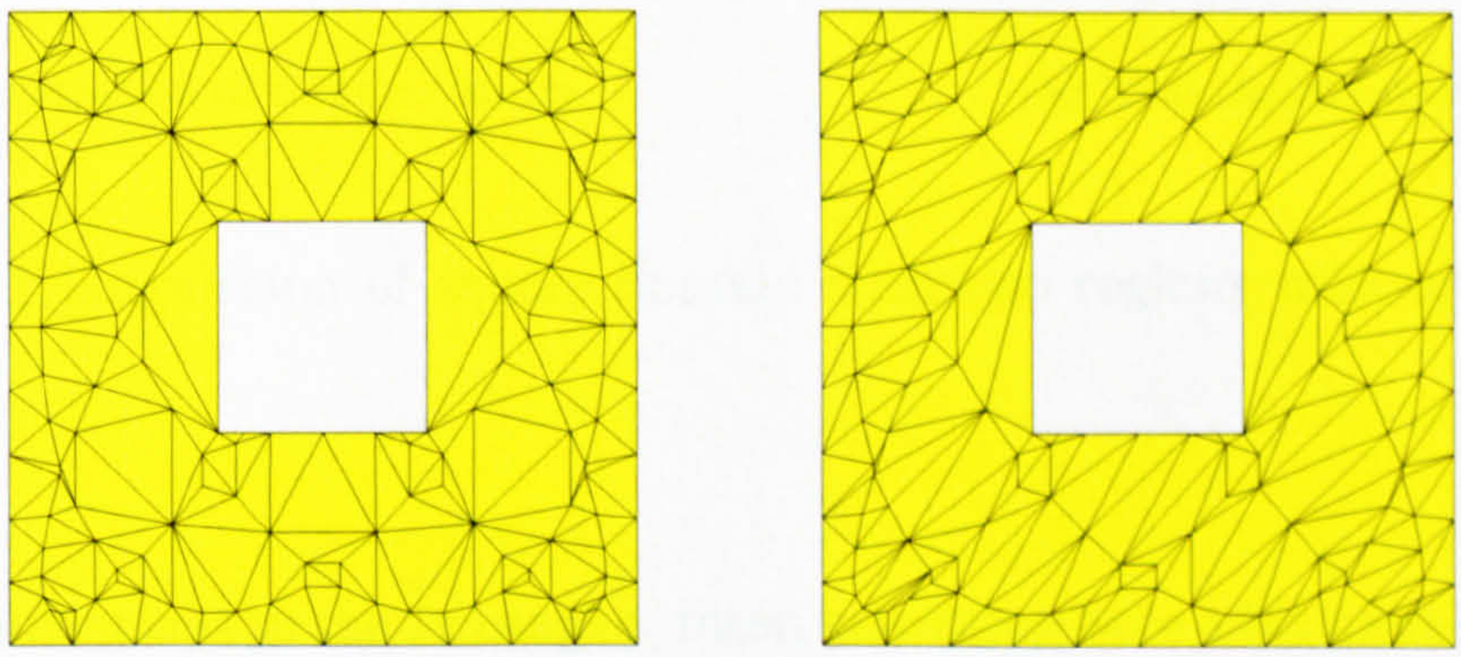


Figure 9.5: Triangulation input: square domain with a hole filled with points from the discretised Lissajous knot.



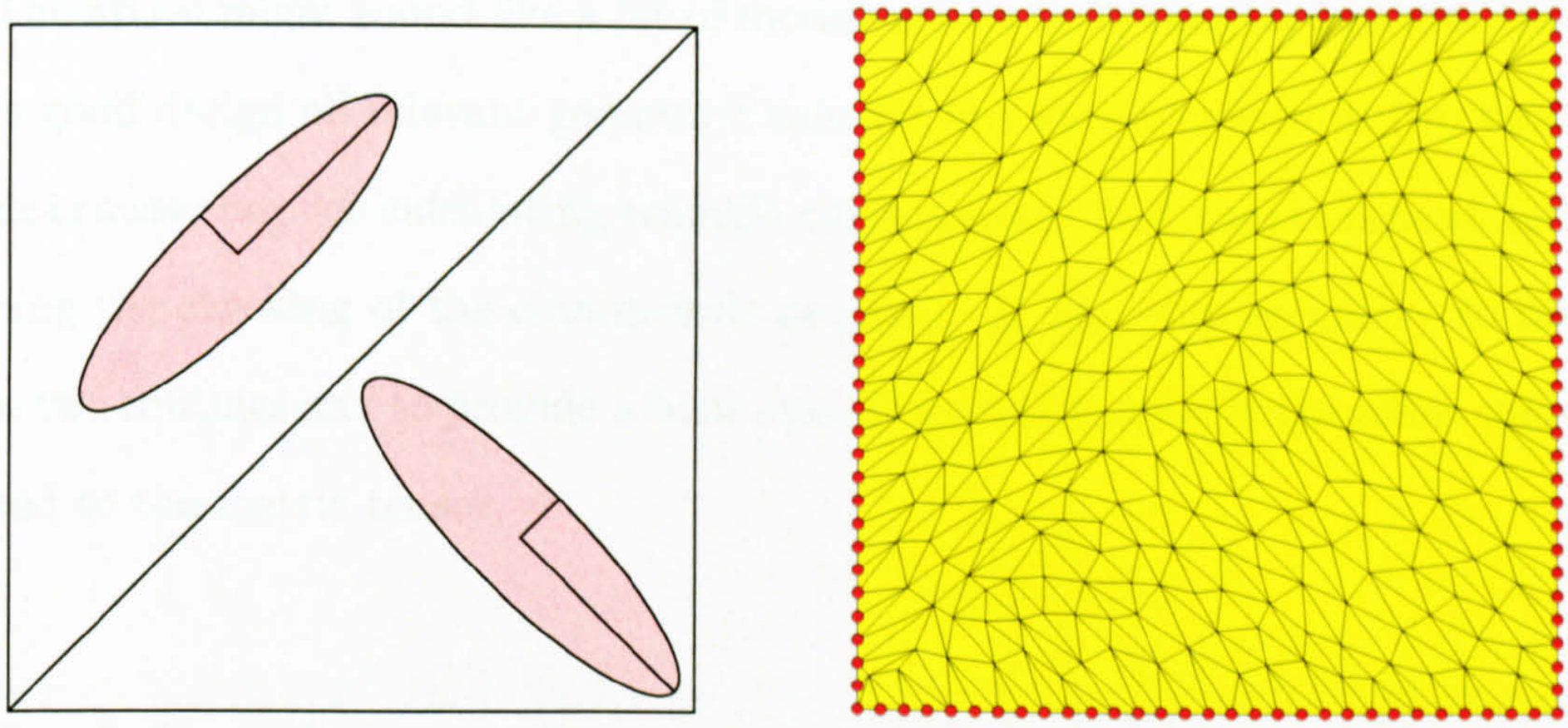
(a) Isotropic (b) Anisotropic

Figure 9.6: Triangulation of domain showed in figure 9.5.

One can think of mesh anisotropy as distortion applied to coordinate points plane treated as a rubber sheet. That distortion can be described by defining a metric tensor field. To make it intuitive, the metric tensor will be described by its eigenvalues and eigenvectors. The metric tensor can be also identified with a local affine transformation which maps circles into ellipses as depicted in Figure 9.3.

The original `Triangle` mesh generator does not provide hooks for defining a local metric tensor, but it was found to be relatively easy to add them. Again the `TriangleGeneratorPro` class was modified, this time adding another parameter which delivers metric tensors. Hence now it is possible to control the triangle orientation and elongation in a very flexible manner. Figure 9.3 shows an anisotropic

mesh for a square domain divided into two regions each with its own constant metric tensor.



(a) Illustration of metric tensors

(b) Two regions mesh

Figure 9.7: Triangulation of square domain with two regions with different metric tensors.

The modifications to the `Triangle` mesh generator were a bit more involved than it could be suggested from the above description. The `Triangle` generator uses the Delaunay algorithm which is based on an empty circumcircle property.[1] If the metric tensor is not a Cartesian one, then the empty circumcircle criterion turns into an empty circum-ellipse criterion. However, checking if a given point falls into a triangle circumcircle is a fairly easy test[2], it is not the same for a triangle circumellipse. This is the reason why the following trick is used: having a metric tensor or effectively a transformation from an undistorted space to distorted one, a reverse mapping is calculated (in the general case it is a local reverse mapping at a given point). Then point and triangle circumc-ellipse are transformed by this mapping which moves the circum-ellipse back to a circle again. Because the transformation is affine then the

[1]This property states that for each element of the triangulation, the triangle's circumcircle does not contain any other vertices than its own.

[2]Easy with arbitrary precision arithmetic. With computer implementation one has however be very careful, otherwise numerical errors can cause erratic behaviour of the test routine.

relation between point and ellipse does not change. However, by doing this one can now use the standard circumcircle test.

The above might sound like a lot of modifications to `Triangle`, however because of its good design all relevant geometric calculations were localised in two routines: `findcircumecter` for calculating triangle circumcenter, and `incircletest` for performing the checking of the circumcircle property. It was only necessary to modify these two routines and to provide a hook enabling users to specify the affine mapping related to the metric tensor.

9.4 Mapping surface properties onto a parametric mesh

Having endowed `Triangle` with the ability to generate nonuniform anisotropic meshes it is possible to build a surface mesh generator. The scheme of the presented mesh generator is based on algorithms described in [74] and includes the following steps:

- Discretization of the mesh boundaries in parametric space. In the case of trimmed NURBS surfaces the trimming curves are discretised using adaptive sampling of parametric curves described in section 6.4.1.

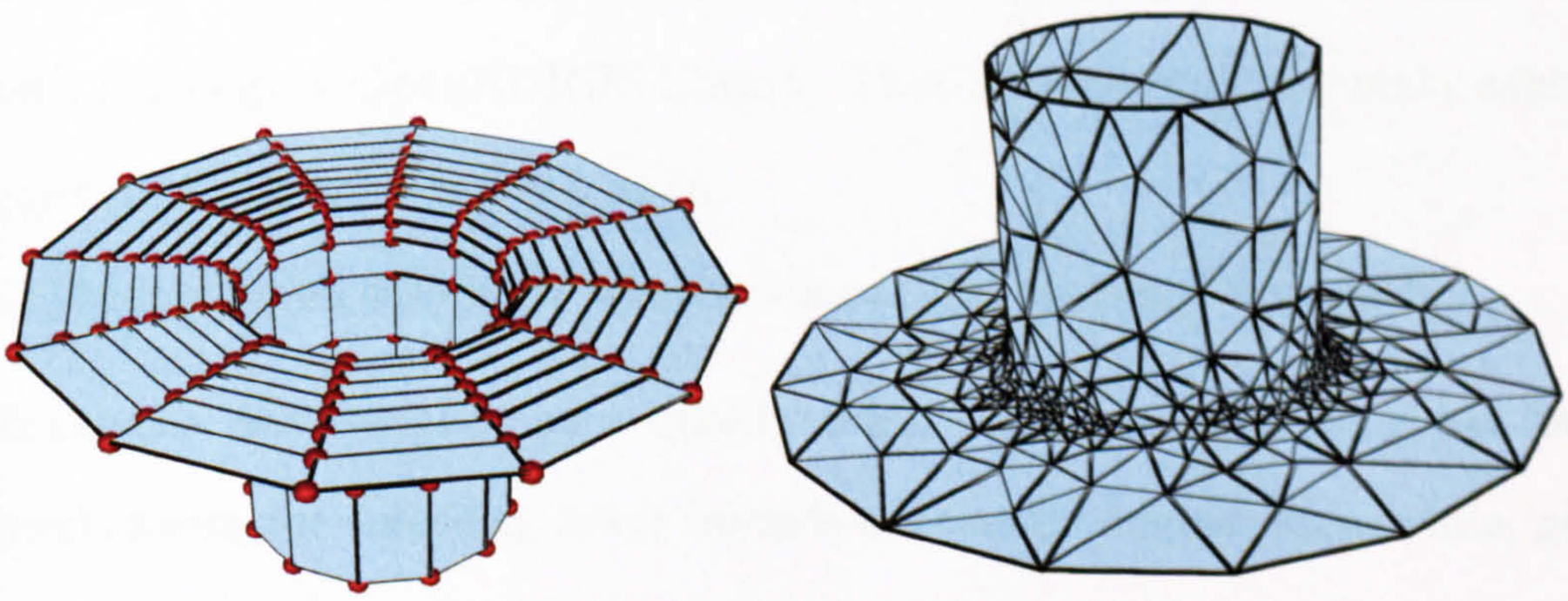
- Creation of initial triangulation from all points on the boundaries in parametric space is formed.

- The initial triangulation is refined. A triangle in parametric space is scheduled for refinement, if for any edge, a segment obtained by connecting the edge endpoints mapped on the surface, violates the curve resolution conditions, supposing the edge is also mapped onto the surface. Here the code used for adaptive curve sampling, presented in section 6.4.1 was reused.

- For each new point in parametric space scheduled for insertion into triangulation, its corresponding point on the surface is calculated. At that point the surface principal stretches and principal stretch directions are calculated as described in section 6.4.2. These principal stretches and directions are treated as elements of the metric tensor and are used to calculate the local affine transformation for the incircle mapping.

- When all triangles satisfy the size criteria then the parametric mesh is mapped onto the surface.

Examples of the application of the above procedure are shown in Figures 9.8, 9.9 and 9.10.

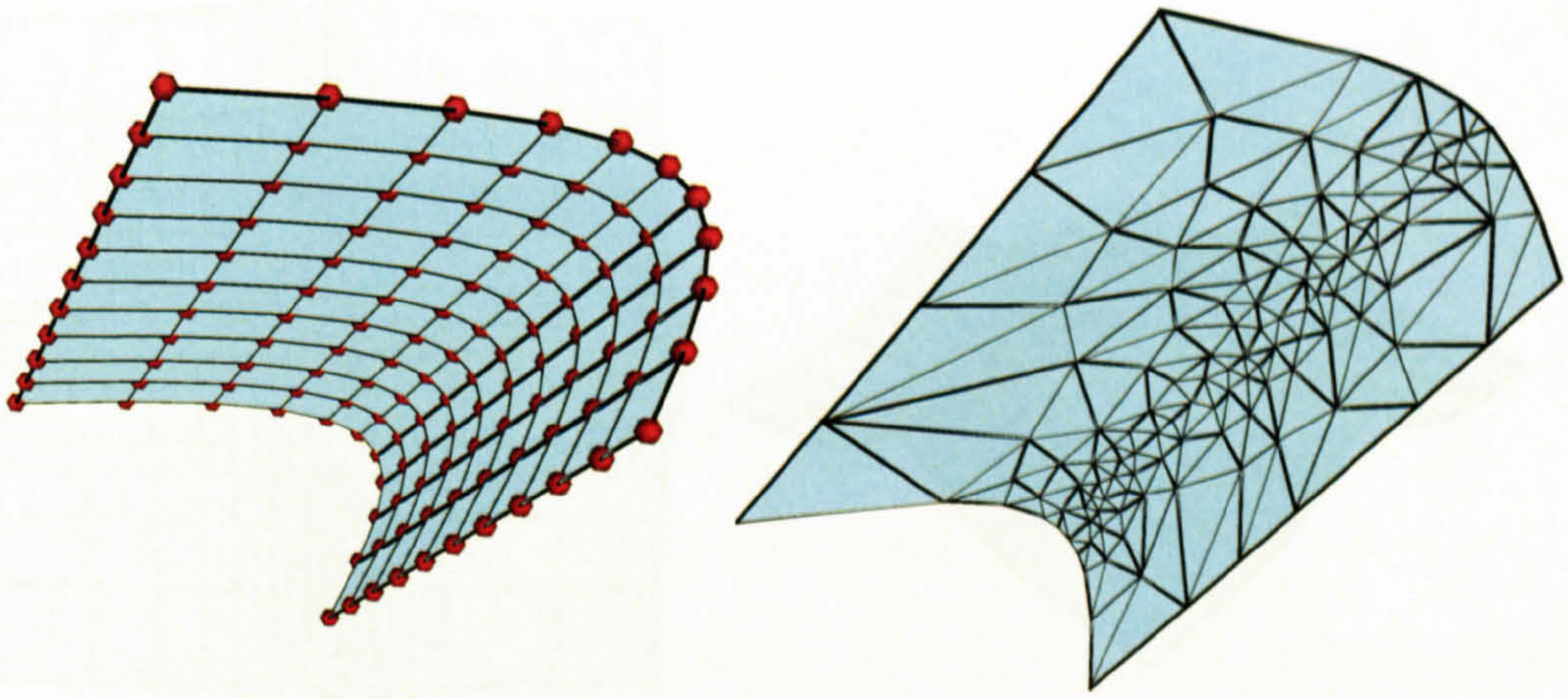


(a) Rotational surface, (b) and its triangulation.

Figure 9.8: Triangulation of rotational NURBS surface.

9.5 Further development

The above case study showed that by selecting appropriate components it is possible to relatively easily construct a surface mesh generator, at least in a basic form. Though the `Triangle` mesh generator was not designed for such tweaking, modify-



(a) Ruled surface, (b) and its triangulation.

Figure 9.9: Triangulation of ruled NURBS surface.

ing it turned out to be not too difficult. All geometry and surface calculations were done with the help of OpenNURBS *library. That allowed only* the main aspects of the algorithm to be concentrated upon.

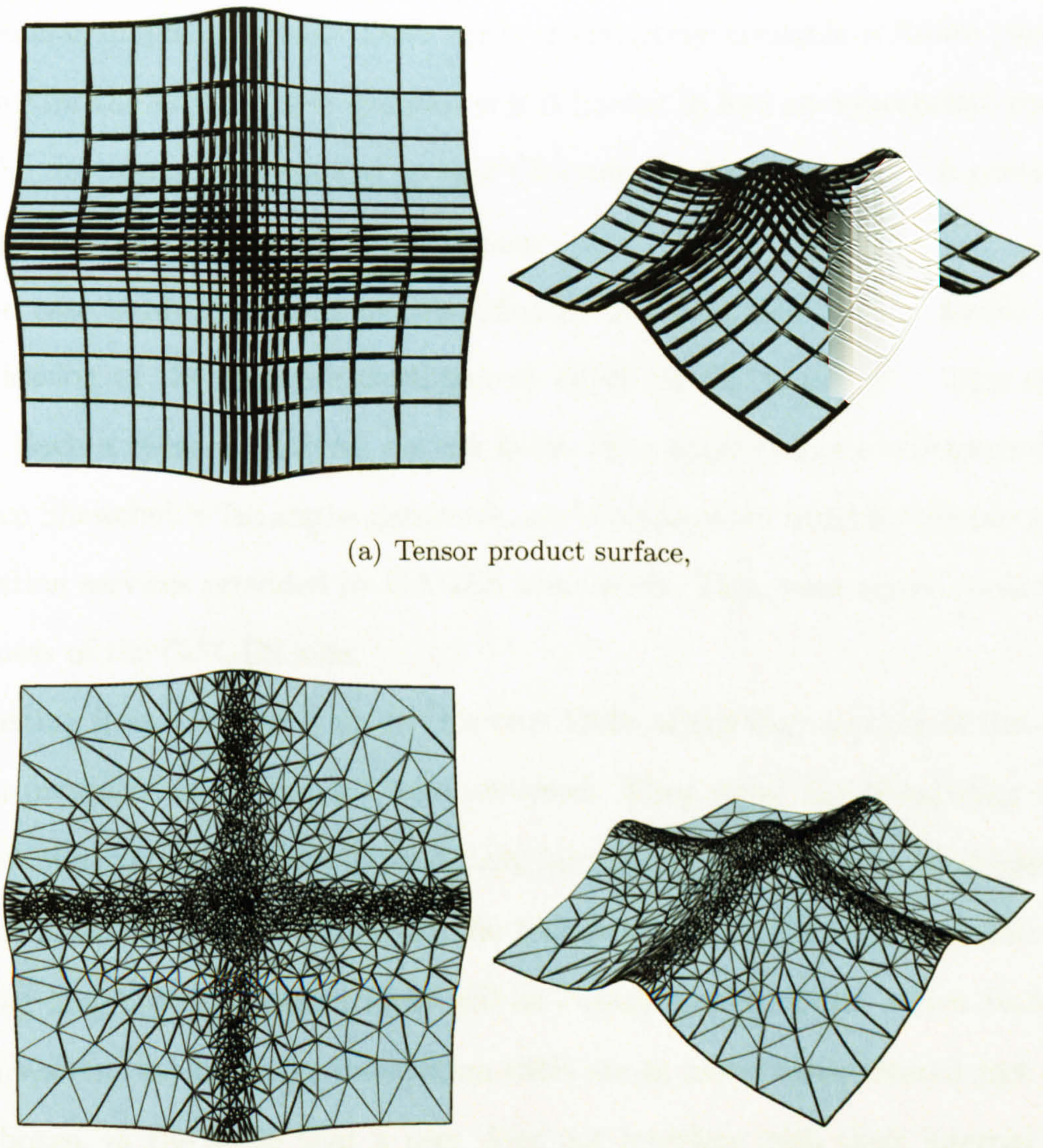
This study showed also some weak points of the proposed approach:

- `Triangle` was designed as a quality mesh generator, that is, it has built-in mechanisms for enforcing lower bounds of triangle angles. Meanwhile, generation of anisotropic meshes may require to keep very bad triangles in parametric space. Synchronising the mechanism for quality mesh generation with surface driven mesh control is the weakest point of the solution presented.

- In the solution presented, surface characteristics are calculated at each insertion point. In practice this approach might turn to be inefficient. A better approach is to discretised surface properties using for instance quad-trees.

- Mesh resolution is driven by surface geometric features only. A useful addition would be to allow users to specify custom density constraints.

As an alternative to `Triangle`, one could use the implementation of incremental Delaunay triangulation provided by the GTS library. The advantage of using



(a) Tensor product surface,

(b) and its triangulation.

Figure 9.10: Triangulation of the tensor product NURBS surface.

GTS is that is simplifies the implementation of edge driven refinement algorithms, thus avoiding problems with the modification of the circumcircle test and related functions.

9.6 Concluding remarks

In this Chapter the development of the surface mesh generator was presented. As indicated in section 9.1 surface meshing is in many approaches the necessary pre-

requisite for volume meshing. In the case of generation of two dimensional or three dimensional simplicial meshes there exists many freely available software packages. However for the surface mesh generation it is harder to find an appropriate package, thus the development presented in this Chapter can be helpful to the researchers dealing with the surface meshing problems.

The case study presented in this Chapter is also a very good example of the minimisation of the software development effort by the reuse of existing components. Such a reuse is enabled, on one hand, by a good design of components, for instance Shewchuk's `Triangle` generator, and on the other hand by the component integration services provided by GAGES framework. This, once again, confirms the usefulness of the GAGES idea.

Starting from Chapter 5 up to this one, three of the four aspects of the hybrid system presented in section 4.3 were analysed. They were: the underlying system programming language libraries, scripting languages, and scripting language interface modules. In the next Chapter the fourth aspect of the hybrid system – the scripting language integration tools will be closely looked upon. When building a hybrid system, the language integration tools are in many cases treated just as the black boxes, in the sense that a user does not interfere with their internal working. However it will be shown in the next Chapter, that sometime it is desirable to customise such tools in order to meet some special needs. The next Chapter will show how to perform such customisations and will also present some new software integration ideas resulting from the close scrutiny of these tools.

Chapter 10

Towards a generic methodology

Previous Chapters showed why scripting components are important and how to use existing tools to extend or embed scripting languages. This Chapter investigates the issue of developing new software integration tools. It discusses why good knowledge of software tools is important and describes the development and application of some of them. As the main test case for this Chapter, the development of the SWIG module for the Ch [93] language was selected. Beside providing a detailed account of the mapping of the C/C++ features to the Ch language, this Chapter describes motives for building new tools, delves into the notion of "SWIG as universal Scientific Interface Definition Language" and presents several practical examples.

10.1 Motivations

Not every programmer needs to know the details of how a compiler works or be able to build a compiler himself, however such knowledge is definitely useful. This applies also to the knowledge of the internal details of the software integration tools. First of all, knowing exactly how the tools work can help in their everyday use, but can also result in their application in quite unexpected areas. Software integration tools are in most cases compiler-like tools performing often quite complex analysis of source code. The results of such an analysis can be used to develop code wrappers

for other languages, but also to build specialised applications within the domain of a single language. A couple of examples of such usage are given at the end of this Chapter.

There is also another issue related to the building of tools, though most of the users will not encounter it. It is natural that programming languages evolve, new languages appear and some fade into oblivion. Though in the mainstream the evolution is quite slow however in the area of domain specific languages it might not be so, as such languages are developed as the needs arise. The ability to quickly develop software integration tools for such languages is valuable as it:

- speeds up new language development, as part of it can be provided as extension modules,
- allows resource intensive operations to be delegated to another layer, for instance implemented in C and compiled for speed,
- allows easier integration with existing components.

In some cases, e.g. for data format converters, software integration tools can be built as simple shell, Sed, AWK, Python or Perl scripts. Other situations, for instance the integration of programming components, might require the development or modification of more complex programs. When developing from scratch, one can use general compiler writing tools such as `flex/bison` [121], `eli` [139] or `antlr` [140]. When modifying an existing program to suit new needs, one has to carefully consider different possibilities to make sure, that the required changes are possible, and that the whole cost of building a new tool are not much higher than the software integration without it. Of course it takes a good knowledge and experience, firstly to pick the right program, and then to evaluate the costs of its modification, beforehand. Testimonials from previous projects, like the one presented in this Chapter for the SWIG modification, are very valuable as they permit learning from somebody's else experience and mistakes.

10.2 Extending SWIG for Ch

10.2.1 Why SWIG

For the case study presented in this Chapter SWIG was chosen as the base tool. SWIG features were already discussed in section 4.3.2 so here only the ones which matters the most for this Chapter will be recalled: namely SWIG extensibility, good documentation and community support. SWIG was designed to support one-to-many language interfacing, and implemented in such way as to make adding new language modules simple. SWIG has complete and detailed documentation including the developers documentation and also an active users community. With all this familiarisation with SWIG source code was relatively easy.

The reasons for rejecting other candidates for the base tool are given below:

- Sip borrowed many ideas from SWIG but was geared specifically towards integrating Python and C++, and especially for wrapping the Qt library. This candidate was rejected because at the time when the author was experimenting with booth tools, Sip documentation was much more sparse than that of SWIG. Also the SWIG implementation seemed to be more open and flexible.

- Boost.Python – Boost.Python is a very strong competitor to SWIG in the area of interfacing Python with C++. However Boost.Python heavily relies on C++ templates and a first look at the source code was for author a little discouraging.

- Babel – this tool was rejected for a similar reason as Boost.Python – on the first sight it looked far to complex to quickly bring profits for the effort invested in learning it.

It should be noted that the choice of SWIG as a tool for this study was done both on the basis of first impressions and a careful technical comparison. Quite important

was also the fact that the author has a couple of years of experience as regular SWIG user, thus for other users the gains from the candidates may vary.

10.2.2 Why Ch

A very succinct characterisation of Ch can be given by citing "The Ch Language Environment – SDK User's Guide, version 5.1" [92]:

Ch is an embeddable C/C++ interpreter. Ch supports all features in the ISO 1990 C standard, most new features in the latest ISO C99 standard including complex numbers and variable length arrays, classes in POSIX, C++, Win32, X/Motif, OpenGL, GTK+, ODBC, WinSock, very high-level shell programming, cross-platform Internet computing in safe Ch, computational arrays for linear algebra and matrix computations, high-level 2D/3D plotting and numerical computing such as differential equation solving, integration, Fourier analysis. Ch can also be used as a login Unix command shell and for high-level scripting such as shell programming, to automate tasks and a common gateway interface in a Web server in both Unix and Windows.

Ch was chosen for the study presented in this Chapter not only because it appears to be very suitable language for scientific computing but also because of the following reasons:

- Ch lacks a specialised support for interfacing with other languages. Though the Ch distribution provides tool called `c2ch` which automates the writing of Ch interface code to C functions, it only works with restricted set of function declarations, for instance it does not support functions with an argument of pointers to functions. Wrapping more complex functions as well as other syntactic constructs (structures, unions, enums) requires manual code writing.

- Ch differs from the languages supported by SWIG either by general language features or by specific details of the API. While Ch is a superset of C and a subset of C++ it does not mean that it was easier to construct the SWIG module for it. While most of the languages supported by SWIG (Java is the

exception) are dynamically typed Ch is statically typed. The Ch interface to C/C++ consists of two parts - one in Ch and the other in C. Finally parameters from the Ch function are passed to the C wrapper using a standard C mechanism for variadic functions (functions with unspecified number of arguments). All this adds to the complexity of the Ch SWIG module but also makes Ch a good case for studying the SWIG implementation.

10.2.3 An anatomy of the Ch extension

Building Ch extensions is described in detail in [92] so here only a basic outline will be given.

The Ch language extension consist in general of three elements:

- The Ch header file, containing a declaration of objects provided by the extension,
- The Ch function file, responsible for binding a function name to an address via calls to a `dl` family of functions for dynamic loading, and for passing arguments to a dynamically loadable C wrapper function,
- The C source file, providing a wrapper implementation, which in turn translates between the Ch and C arguments and calls the extension function.

In the case of wrapping a single C/C++ function it is possible to merge the header file and Ch function file, but here a more general setting for dealing with multi-function/multi-class extensions will be kept.

Ch function files have names with a suffix `.chf`, thus we will use the name "chf function" to denote that part of the Ch interface. C wrappers in turn are named with suffix `_chdl`, thus "chdl functions" will be discussed.

In the case of wrapping most of the functions, all three interface components can be derived just from a function declaration. Special cases however, like overloaded

functions, classes, etc, may require special SWIG directives to guide the wrapping process.

To make the presentation more substantial wrapping of a simple function with the following signature will be considered:

```
int is_positive(double val);
```

Listing 10.1: A simple C function to be wrapped in Ch.

Additionally the extension module will be assumed to be called `numbers`.

The interface header file

The Ch interface header file for the sample extension is shown in listing 10.2

```
1  #ifndef NUMBERS_H
2  #define NUMBERS_H
3  #if defined(_CH) || defined(_CH_)
4  #include <stdio.h>
5  #include <dlfcn.h>
6  #include <stdarg.h>
7  void *g_Chnumbers_handle = dlopen("libnumbers.dl", RTLD_LAZY);
8  if(g_Chnumbers_handle == NULL) {
9      fprintf(_stderr, "Error: dlopen(): g_Chnumbers_handle\n",dlerror());
10     fprintf(_stderr, "cannot get g_Chnumbers_handle in numbers.h\n");
11     exit(-1);
12 }
13 void _dlclose_numbers(void) {
14     dlclose(g_Chnumbers_handle);
15 }
16 atexit(_dlclose_numbers);
17 extern int is_positive(double arg1);
18 #endif /* defined(_CH) || defined(_CH_) */
19 #endif /* NUMBERS_H */
```

Listing 10.2: The Ch interface header file.

The role of this header file is to declare a handle to a dynamically loadable library (*DLL*) containing a C wrapper and to load that library through a call to the `dlopen()` function, see line 7. It also declares an extension function for the Ch space in line 17, and arranges some cleanup after the extension has been used, lines 13–16.

The interface Ch function

The Ch function part of the interface is shown in listing 10.3

```
1  int is_positive(double arg1) {
2    void *fptr;
3    int retval;
4    fptr = dlsym(g_Chnumbers_handle, "is_positive_chdl");
5    if(fptr == NULL) {
6      const char* msg = dlerror();
7      fprintf(_stderr, "Error: %s(): dlsym(): %s\n",__func__, msg);
8      return -1;
9    }
10   dlrunfun(fptr, &retval, NULL,  arg1);
11   return retval;
12 }
```

Listing 10.3: The interface Ch function.

The role of this function is to fetch the address of the C wrapper function from a dynamically loadable library, line 4, and to call that wrapper function passing to it arguments from the Ch space, line 10. In line 10 the address of variable used for holding the result value is also passed to the wrapper.

The interface C function

Finally the last part of the interface is shown in listing 10.4.

```
1  #include <ch.h>
2  EXPORTCH int is_positive_chdl(void *varg) {
3    ChInterp_t interp;
4    ChVaList_t ap;
5    double arg1 ;
6    int result;
7
8    Ch_VaStart(interp, ap, varg);
9    arg1 = (double) Ch_VaArg(interp, ap, double);
10   Ch_VaEnd();
11   result = (int)is_positive(arg1);
12   return result;
13 }
```

Listing 10.4: The interface C function.

The C wrapper function obtains as a single argument an opaque void pointer. From this pointer the real arguments are extracted using an interface similar to the one used for the variadic functions in C, lines 8, 9, 10. After obtaining the arguments the wrapped function is called in line 11.

It should be noted that the above example presents a complete though simplified view of the Ch interface. In the case of arguments of arrays, strings, pointers to functions, structures and respectively similar return types, the interface becomes more complicated. It is even more complex when considering the wrapping code implemented in C++.

10.2.4 The anatomy of a SWIG module

SWIG is a significantly complex tool, performing complicated analysis and transformation of C/C++ source code. Generally, SWIG can be logically divided into a front-end, a back-end and language modules. The role of the front-end is to parse C/C++ code together with special SWIG directives and to build a parse tree augmented with additional information necessary for building wrappers. The role of the back-end is to traverse that tree and to call for each tree node handler function defined in the language module. Language modules provide handler function implementations, specific to each output language supported by SWIG.

This organisation of SWIG, and particularly the encapsulation of language specific code in modules, allows the extension of SWIG in a clean manner. For module writers SWIG offers a consistent and clean API for:

- handling primitive data structures (strings, lists, hashes),
- handling I/O operations,
- navigating and manipulating parse trees,
- working with C/C++ types,
- handling function parameters.

Each concrete language module is implemented as a class derived from the Language base class. The Language class provides methods which are handlers of a particular type of parse tree node. Most of the functions are declared as `virtual` and have to be overridden in derived classes. Listing 10.5 give a glimpse of partial declaration of the Language class.

```
Language();
virtual ~Language();
virtual int emit_one(Node *n);
/* Parse command line options */
virtual void main(int argc, char *argv[]);
/* Top of the parse tree */
virtual int  top(Node *n);
/* SWIG directives */
virtual int applyDirective(Node *n);
virtual int clearDirective(Node *n);
virtual int constantDirective(Node *n);
virtual int extendDirective(Node *n);
virtual int fragmentDirective(Node *n);
virtual int importDirective(Node *n);
virtual int includeDirective(Node *n);
virtual int insertDirective(Node *n);
virtual int moduleDirective(Node *n);
virtual int nativeDirective(Node *n);
virtual int pragmaDirective(Node *n);
virtual int typemapDirective(Node *n);
virtual int typemapcopyDirective(Node *n);
virtual int typesDirective(Node *n);

/* C/C++ parsing */
virtual int cDeclaration(Node *n);
virtual int externDeclaration(Node *n);
virtual int enumDeclaration(Node *n);
virtual int enumvalueDeclaration(Node *n);
virtual int enumforwardDeclaration(Node *n);
virtual int classDeclaration(Node *n);
virtual int classforwardDeclaration(Node *n);
virtual int constructorDeclaration(Node *n);
virtual int destructorDeclaration(Node *n);
virtual int accessDeclaration(Node *n);
virtual int namespaceDeclaration(Node *n);
virtual int usingDeclaration(Node *n);
```

```
    /* Function handlers */
    virtual int functionHandler(Node *n);
    virtual int globalfunctionHandler(Node *n);
    virtual int memberfunctionHandler(Node *n);
    virtual int staticmemberfunctionHandler(Node *n);
    virtual int callbackfunctionHandler(Node *n);

    /* Variable handlers */
    virtual int variableHandler(Node *n);
    virtual int globalvariableHandler(Node *n);
    virtual int membervariableHandler(Node *n);
    virtual int staticmembervariableHandler(Node *n);

    /* C++ handlers */
    virtual int memberconstantHandler(Node *n);
    virtual int constructorHandler(Node *n);
    virtual int copyconstructorHandler(Node *n);
    virtual int destructorHandler(Node *n);
    virtual int classHandler(Node *n);

    /* Miscellaneous */
    virtual int typedefHandler(Node *n);

    /* Low-level code generation */
    virtual int constantWrapper(Node *n);
    virtual int variableWrapper(Node *n);
    virtual int functionWrapper(Node *n);
    virtual int nativeWrapper(Node *n);
};
```

Listing 10.5: Partial definition of SWIG's Language class.

The most important of the above methods and the central point of each concrete language class is the `functionWrapper()` method. It is important, because all other methods, after applying their specific transformations to parse tree nodes, eventually call the `functionWrapper()`. This function effectively generates wrapper code.

However, it should be noted, that not all wrapper code is generated from scratch by `functionWrapper()`. A lot of SWIG processing and internal configuration is managed not by code written in C but by configuration files in the SWIG library.

The SWIG library also provides the most commonly used functions and wrapper fragments in the form of generic macros. A language module can use these generic templates, but most of the time, some of them are overridden by language specific interface files.

10.2.5 Mapping C/C++ features to Ch

Wrapping ordinary functions

Wrapping of ordinary functions, that is functions which are not overloaded, which do not have a default value, which do not have arguments or return values as pointer to function nor array nor string, does not differ significantly from the scheme shown in section 10.2.3. The only difference is that all chdl functions are declared as returning `int`. The return value is used to check if the function invocation was successful. The pointer to the variable holding the result is passed as the last, additional function argument.

Wrapping overloaded functions

The Ch language does not natively support overloaded functions, however they can be simulated using variadic functions, which are supported very well in Ch. The trick is based on the fact, that variadic function can accept any number of arguments of any type. Inside the variadic function, the number and type of arguments can be checked and the matching overloaded version called. This mechanism is described in [92]. The Ch SWIG module generates several chdl function wrappers, one for each overloaded version. The chf function part of the wrapper works then as a dispatch function – it checks for the number and type of arguments and calls an appropriate chdl wrapper. Checking the number and type of arguments would be more efficient in C space and originally the dispatch function was generated as additional chdl wrapper. However it became apparent, that this solution works only for built-in types. Because of the way types are mapped to integer tags, it is impossible to

distinguish between user defined types (classes or structures) on the C level. Moving the dispatch code to the Ch level yielded maybe a less efficient but a more general solution.

It should be noted that the above support for overloaded functions works with the assumption that all overloaded versions return the same type. If this is not the case then one has to assign a unique name to each function with a different type. It can be conveniently done using default SWIG's `%rename` directive. This directive can be also used to solve the efficiency problem if the overhead induced by the dispatch mechanism in Ch space turns out to be not acceptable.

Wrapping functions with default arguments

C++ functions can have default arguments. Ch does not support this feature but again, it can be simulated using variadic functions. The chf function part of the wrapper checks for the number of arguments and if the argument list is shorter than required, the missing arguments are assigned default values. The chdl part of the wrapper is built as in the case of ordinary functions. In this way default arguments can be declared even when wrapping C functions.

Wrapping variadic functions

Variadic functions cannot be wrapped directly. As described in [92] a variadic function of a signature:

```
void foo(int arg, ...);
```

has to be transformed into two functions:

```
void foo(int arg, ...) {
  va_list ap;
  va_start(ap);
  foo_imp(arg, ap);
  va_end(ap);
}
void foo_impl(int arg, va_list ap) {
  ...
```

```
}
```

With such a rearrangements function `foo_impl()` can be wrapped in a fairly standard way and in Ch it is seen again as `void foo(int arg, ...)`.

Wrapping pointers to functions

Though the Ch and C/C++ extension codes share the same address space, pointers to functions cannot be passed directly between Ch space and C/C++ space. To illustrate the problem it is assumed that the following function to be wrapped expects a callback pointer as its second argument:

```
typedef int (*fptr)(int);
int call_it(int v, fptr callback);
```

The corresponding fragment of the C wrapper code may look as follows:

```
1   typedef int (*fptr)(int); /* function pointer type */
2   static ChInterp_t interp;
3   /* C function to replace the Ch function pointer */
4   static int call_it_chdl_callback(int);
5   /* variable used to save the function pointer from the Ch space */
6   static void *call_it_callback;
7   EXPORTCH int call_it_chdl(void *varg) {
8   ChVaList_t ap;
9   int  arg1;
10  int result;
11  fptr handle_ch, handle_c = NULL;
12  Ch_VaStart(interp, ap, varg);
13  arg1 = Ch_VaArg(interp, ap, int);    /* get 1st argument
14  /* get and save Ch function pointer */
15  handle_ch = Ch_VaArg(interp, ap, fptr);
16  call_it_callback = (void *)handle_ch;
17  if(handle_ch != NULL) {
18    handle_c = (fptr)call_it_chdl_callback;
19  }
20  result = call_it(arg1, handle_c);
21  Ch_VaEnd(interp, ap);
22  return result;
23  }
24  static int call_it_chdl_callback(int arg) {
25  /* Call Ch function by its address */
```

```
26    int res;
27    Ch_CallFuncByAddr(interp, call_it_callback, &res, arg);
28    return res;
29  }
```

From the above code one can see that the pointer to the Ch function (here saved in variable `handle_ch`) is not passed to the function `call_it` but instead the pointer to the C function `call_it_chdl_callback` is passed. The function `call_it_chdl_callback` in turn uses the Ch API function `Ch_CallFuncByAddr` to call the Ch callback function. It should be noted that in the above scheme one has to manage two global variables: one is `inter` for reference to the Ch interpreter and the other is `call_it_callback` storing the Ch function pointer. If the Ch function pointer is consumed immediately, that is if it is called before the C wrapper function returns, then the above scheme works well. However if one wishes to store somewhere the callback pointer for subsequent calls then this scheme will probably crash because of the multiple calls to function `call_it()` which will override the value of global variable.

One possible solution is to have instead of single a callback pointer and a single callback wrapper, an array of callback pointers, corresponding to the number of callback wrapper definitions, and an array of callback wrapper pointers. Then with each call to `call_it` an internal counter will be incremented and new slots from the arrays will be used. In this way successive calls to `call_it()` will not override the stored values. This solution was used in Ch for wrapping some OpenGL functions. Of course for some applications all the array slots may quickly be filled up.

Thus for the purpose of this study and for the Ch SWIG implementation some improvements to the above scheme were proposed.

Again, as in the above scheme, the callback support mechanism relies on the static array of callback pointers. This array will be called the "callback dispatch table". The improvements account for:

- Caching the last callback pointer used. A new slot from the callback dispatch

table is used only if the callback pointer has changed since the last call. This way, having the callback dispatch table of size N, we do not have a maximum of N calls but rather N alternating calls, i.e. calls where the callback pointer changes.

- Searching the callback dispatch table and using new slot only if the callback pointer is not in the table. In this way instead of a maximum N calls we have N calls with a unique value of callback pointer.

Naturally both of the improvements can be used at the same time, saving in this way some search operations.

Of course the search in the callback dispatch table takes some time. In order to minimise the overhead the callback dispatch table is kept sorted using callback pointers as keys. This allows us to use a binary search algorithm when searching for callbacks. The time complexity for a dispatch table of size N is as follows:

- $O(1)$ – this is the case if the callback pointer has not changed from the previous call

- $O(\log_2(K))$ – this is the case when the callback pointer is already in the table. The value K ($K <= N$) is the number of already filled slots of the dispatch table.

- $O(K)$ – this is the case when the callback pointer is not in the table. The linear complexity comes from the fact that the dispatch table needs to be kept sorted and a simple linear insertion scheme was used.

In the last case complexity can be improved by using a different data structure than a sorted table, but this results in a more complex implementation. Taking into account that the dispatch table will not have to many entries it is reasonable to assume that the overhead will not be too big.

All three presented approaches – a single global variable, the greedy use of a callback dispatch table, and searching the table, have their advantages and situations

when one performs better than others. In the current Ch SWIG module the third solution was taken as the default one, but future version of Ch SWIG will allow the user to specify in an interface file which of the three solutions should be used.

Wrapping typedefs

SWIG handles C/C++ types with a full awareness of the `typedef` declarations, thus wrapping them for Ch was quite straightforward. The `typedef` declarations are simply copied verbatim to the Ch interface header file.

Wrapping enums

Declarations of enum types are handled the same way as `typedef` declarations – they are copied verbatim to the Ch interface header file. At the time of writing this thesis one kind of enum as well as typedef declarations is problematic in Ch SWIG, namely the ones placed inside the class scope. At the moment Ch does not support such constructions in all cases.

Handling bool type

Ch does not support `bool` type, however Ch SWIG seamlessly handles arguments and return values of this type by converting them automatically to `char` type.

Wrapping structures

Though C/C++ and Ch share the same address space and pointers to structures can be freely exchanged, it does not mean that the layout of the structure in C/C++ and Ch will be the same. It also does not mean that the assignment to structure elements or retrieval can be done unadorned.

First of all the user may decide not to wrap some data members (they are automatically removed from structure declaration in Ch space).

Additionally the user might want a finer control over read/write access to data

members. Also, C/C++ data structures might be extended with additional fields in a scripting language.

Finally one cannot do straightforward assignment of data members which arc pointers to functions. Ch pointers to functions must be transformed into appropriated C callback wrappers.

The above are the reasons that accessing structure data members are done through special `set/get` methods generated automatically for each data member. It is also assumed that both C data structures and C++ classes are mapped to Ch classes. This allow us to write in Ch

```
1          struct Foo a;
2          a.var_set(23);
```

even if `Foo` is C structure:

```
1          struct Foo {
2              int var;
3          };
```

The business with `set/get` accessors complicates a little the use of wrapped structures, but on the other hand it allows uniform treatment of all data structures, regardless if they have pointers to functions as data members or not.

Wrapping C++ classes

Treatment of C++ classes does not differ from the treatment of C structures. The difference accounts for using `new/delete` operators in the case of C++ and `malloc` family of functions the in case of C. Additionally as Ch only supports some of the C++ features, SWIG issues a warning message and skips problematic declarations if it encounters copy constructor declarations, overloaded operators, friend declarations.

Handling inheritance

Ch does not have the notion of inheritance but at least two ways of handling it can be devised:

- one is to require users to use pointers or references explicitly casted to base classes in order to access methods defined in base classes,
- the other is to copy declarations of all methods from base classes to the class in question.

This first way is the simplest as it requires only declaration of base classes to be provided. It does however break Ch/C++ source code compatibility as the user has to do the explicit casting.

The second way allows the use of C++ code without any changes but on the other hand it can generate class declarations with a huge number of members and huge number of Ch interface functions (big *.chf files). The bigger problem is however the resolution of the inherited methods in the presence of multiple inheritance and especially diamond shape inheritance graphs. In C++ this is solved by virtual inheritance, however solving this problem in the Ch SWIG module will substantially complicate the Ch SWIG code.

In the present implementation a Solomon solution has been chosen: only methods from the direct base classes are copied to the class in question. To access other methods one has to use explicit casting.

Handling C++ templates

SWIG fully supports C++ templates syntax, however in order to be wrapped they have to be explicitly instantiated for each type that they are going to be used with. Concrete instances of templates are treated in the same way as ordinary functions and classes.

10.3 Examples of the Ch SWIG usage

The working of the Ch SWIG module was tested on several C and C++ libraries. The Ch interfaces to those libraries were built as part of CHASE project (CH Applications for Scientific Environments) [147]. Below a short description of these libraries is given. The source code for these Ch interfaces as well as some examples and documentation can be viewed and downloaded from CHASE web page [147].

ChLibplot – Ch interface to GNU Libplot library [134]. Libplot is a function library for a device-independent two-dimensional vector graphics library. Libplot is a part of a bigger package GNU Plotutils. Libplot provides C and C++ bindings but in this test case only wrappers for the C API were built. Wrapping Libplot tested the basic Ch SWIG module functionality in case of C code.

ChFLTK – Ch interface to FLTK [141]. FLTK stands for Fast Light Tool Kit and it is a C++ Graphical User Interface toolkit for UNIX, Microsoft Windows and MacOS. Wrapping the FLTK library was done in order to test basic Ch SWIG module functionality in the case of the C++ code.

ChCDT – The Ch interface to the CDT (Container Data Types) library. CDT is a C library written by Vo [148] and it provides a uniform set of operations to manage dictionaries based on common storage methods: list, stack, queue, ordered set/multiset and unordered set/multiset. Wrapping this library permits the testing of Ch SWIG for a more complex C library, and particularly enabled pointers to functions and structures handling to be tested.

ChAgraph – The Ch interface to Agraph library [142]. Agraph is a C library that defines data types and operations for graphs comprised of attributed nodes, edges, and subgraphs. Agraph is a part of the bigger Graphviz package [143].

ChPLplot – Ch interface to PLplot library [144]. Though the Ch Professional

Edition (which is free for academic use) is shipped with a package for functions and data plotting, wrapping the PLplot library provides an alternative solution which can be also used with the Ch Standard Edition. Wrapping this library with Ch SWIG showed that it is a very handy tool, requiring for the C case very minimal user additions to the interface file, and that building a complete interface even to a complex C library can be done in couple of hours.

ChGnuplot – Ch interface to `gnuplot` program [145],[146]. This is another plotting solution for Ch, this time based on `gnuplot` program. It is based on the `gnuplot_i` interface to `gnuplot` by Devillard [145].

10.4 Integration of Ch and Python

So far we have considered writing Ch interfaces to C/C++ code. However it would be also very practical to be able to extend Ch in other languages, for instance Python, or to embed Ch in them. In this section a simplified solution for extending Ch with Python will be considered and then we will indicate how experience gained while building the Ch SWIG module can be used to provide a more general solution.

Calling Python functions from Ch space requires importing a Python function from an appropriate module, translating a Ch argument list into a corresponding Python list calling a Python function and then translating the results back to Ch. Most of the difficulties lie in translating the Ch variables to Python objects and vice versa, as Ch and Python have only a partially overlapping set of types. Full translation between an argument list in Ch and Python is application dependent and requires an extra information which cannot be expressed purely within Ch or Python syntax only. For instance the Ch argument of the signature `int *m` may denote an array of integers or a scalar input or output variable passed by a pointer.

Thus for the most general solution one has to resort to interfaces which are built at compile time on the basis of extra information concerning how to treat Python

function arguments and return value. However, if one restricts oneself to the types common to both languages and some special cases of other types, then it is possible to build a run time support for calling Python functions from Ch. While in principle restricted, the presented solution can handle enough practical cases to make it useful.

The chpython library

The whole support for calling Python from Ch is built around a very simple library consisting of the following functions

```
void ChPy_Initialize(void);
void ChPy_Finalize(void);
void* ChPy_ImportFunction(const char *module, const char* function);
void* ChPy_Call(void *callback, va_list ap);
int ChPy_GetIntResult(void *result, int narg);
int ChPy_NumberOfValues(void *result);
double ChPy_GetDoubleResult(void *result, int narg);
char * ChPy_GetStringResult(void *result, int narg);
```

Listing 10.6: The API of the chpython library.

The functions `ChPy_Initialize()` and `ChPy_Finalize()` initialise and finalise the Python interpreter. The function `ChPy_ImportFunctions()` imports callable objects from a given module and returns the pointer to the corresponding `PyObject` casted as the void pointer. Function `ChPy_Call()` is in turn used to invoke callable `PyObject` passed to it through this void pointer as the first argument. It returns `PyObject` holding result value, again casted to void pointer. Finally functions `ChPy_NumberOfValues()`, `ChPy_GetDoubleResult()`, `ChPy_GetIntResult()`, and `ChPy_GetStringResult()` are used to manage return value from Python functions, returning respectively number of return values, and i-th return value as double, int or string type.

Python function proxy

The above functions are wrapped in Ch and can be used to manage calls to a Python function. However, to make the interface more convenient and safer, a

class `PyFunctionProxy` was added in the Ch space. This class hides from the user management of the callable Python object and the result returned by it. A callable python object is automatically created in the `PyFunctionProxy` constructor and discarded in its destructor. The object holding the result of the call to the Python function is kept as private inside the `PyFunctionProxy` and respectively discarded when not in use – either before a new call to the Python function or when destroying the proxy object.

The declaration of the `PyFunctionProxy` is shown in listing 10.7

```
class PyFunctionProxy {
  public:
    PyFunctionProxy(const char *module, const char *name);
    ~PyFunctionProxy();
    int Call(...);
    const char* GetName(void);
    const char* GetModule(void);
    int GetIntResult(int index);
    double GetDoubleResult(int index);
    char * GetStringResult(int index);
    int NumberOfValues(void);
    void DiscardResult(void);
  private:
    const char *name;
    const char *module;
    void *callback;
    void *result;
    void error_handler(char *message);
};
```

Listing 10.7: Declaration of the PyFunctionProxy class.

With the `PyFunctionProxy` class, calling a Python function is really simple. An example showing the use of a routine from the standard Python module `colorys` for translating between RGB and HSV colour spaces is shown in listing 10.8

```
#include "PyFunctionProxy.h"

int main() {
  void *callback;
  class PyFunctionProxy rgb_to_hsv = PyFunctionProxy("colorsys", "rgb_to_hsv");
```

```
  rgb_to_hsv.Call(1.0, 0.0, 0.0);
  printf("Hue %f\n", rgb_to_hsv.GetDoubleResult(0));
  printf("Saturation %d\n", rgb_to_hsv.GetDoubleResult(1));
  printf("Value: %d\n", rgb_to_hsv.GetDoubleResult(2));
  return 0;
}
```

Listing 10.8: Sample program using the `PyFunctionProxy` class to manage the call to the Python function.

More complex cases and automatic solution

The `chpython` library and the proxy class can be easily extended to handle other fundamental types (short, long, etc) and arrays of them. However, this solution does not work for callback pointers, structures and classes. For each kind of these types customised handling code must be written. With a lot of such types writing that code by hand is both tedious and error prone. It is however possible to modify SWIG and to generate such code automatically. To do it, two things have to be done. First of all a Python function will have to be described in terms of the C/C++ syntax plus optional SWIG directives, and a SWIG module must be provided that will not generate calls to the C/C++ routine but will use the Python API to forward the call to the Python interpreter. Though in practice slightly more difficult, the new module will be a merging of SWIG Ch and Python modules. One can imagine also other merges - Ch and Tcl, Ch and Lua, etc. Generalisation of this approach leads to the concept of the SWIG interface as a universal Interface Description Language described in next section.

10.5 SWIG interface as a universal Scientific Interface Description Language

SWIG was designed as a one-to-many integration tool, that is a a tool to build interfaces between C/C++ and other languages. As such, SWIG takes as a basis

the C/C++ header files augmented with special directives.

In the case of building many-to-many integration tools one has to introduce a special interface definition language from which bindings to specific languages are generated. Tools like Babel or CORBA, each define its own interface definition language (IDL).

One can of course argue what properties make an IDL universal and what make it suitable for scientific applications, however from a very utilitarian point of view one thing is certain – specifications in such an IDL should be easy to translate to the target languages. Only then can an IDL gain popularity and be commonly used. In other words it means that there should be an appropriate IDL parser and an extensible code generator.

There are basically two main approaches when building a multi-language integration tool. One is to design an IDL first and then to build a parser and code generator for it, and the other is to start with an existing parser for some language and augment it with the code generation modules for the target languages.

The first approach appears an appropriate strategy because the IDL can be shaped to whatever needs are required, and the approach is not restricted by a given syntax. However, in author's opinion this is also a much harder strategy as we have to design an IDL and build a parser for it. Of course one can argue, that with present compiler technology it is relatively easy to devise a new XML language and generate a parser for it. But unless someone has experience and is fluent in computer languages, it is very likely that the design and implementation will have some inadequacies and it will need time to mature and eliminate bugs. It is not definitely an approach to be recommended for a single developer or a small research group expecting to have not an ideal but a practical tool within a reasonable time.

On the other hand if an available parser for some language is taken, then though restricted by the given syntax and implementation, a working tool will be available from the beginning. Its deficiencies and dark corners should be known, and it will

be much better to deal with known problems than with the unknown ones.

The above rather obvious observations gave birth to an idea of changing the character of SWIG from a one-to-many tool to a many-to-many tool, and to use the SWIG interface specification as a universal Scientific Interface Definition Language. The idea is really simple. Currently SWIG takes the C/C++ interface specification and generates a wrapper for a target language. However, nothing stops us treating the C/C++ interface not as a concrete one (i.e. having an implementation) but as an abstract one, and to generate interface code for host and extension language though not necessarily for C/C++, but for instance Tcl and Python.

There are several advantages to this approach. Beside the one mentioned above, a working solution is available from the start, the others are:

- SWIG was designed having extensibility in mind so many problems of hacking a third party code are avoided,

- SWIG was already ported to many platforms and is quite stable which again minimises the development effort,

- The IDL will be a superset of C++, but in many cases pure C or C++ syntax will suffice. This saves users the need to learn one more language, makes starting using the tool is very easy, builds tool popularity on the basis of the spread and understanding of C/C++.

- Most of the high level languages provide C APIs for interfacing with them. Choosing C/C++ as the basis for the IDL will simplify writing the code generators.

- Expressing the interface in augmented C/C++ syntax will yield more concise and easier to understand descriptions than a solution based on XML format because the C/C++ syntax is closer to the syntax of many languages than XML syntax.

One may wonder if C++ syntax is flexible enough to allow expressing all features required for an universal IDL. The definitive answer will be not given here as this warrants further study, but on the first sight it looks like the answer will be positive, especially taking into account such syntactic constructs as templates or namespaces. Additionally for things which can not be forced into C/C++ syntax, SWIG directives are available. Especially two of them, `%feature` and `%typemaps` are worth noting as they will allow the IDL to be shaped in almost any desired way. Another thing which helps to turn the SWIG interface file into an universal IDL is the way the SWIG parser treats unknown types. The SWIG parser checks the interface file only to ensure syntactic correctness, leaving semantic analysis of declarations to code generation modules.

As a very simple example of the proposed approach the SWIG based SIDL specification for Python function `rgb_to_hsv()` used in listing 10.8 may look as shown in listing 10.9.

```
%interface colorsys
touple<int, 3> rgb_to_hsv(touple<int, 3> INARG);
```

Listing 10.9: Example of a SWIG based SIDL specification for the Python function `rgb_to_hsv()` from the `colorsys` module.

10.6 Miscellaneous applications

As said in the introduction to this Chapter such a flexible tool such as SWIG can be used not only to generate interfaces between various languages, but can also be used to automate single language application building. Here are four possible ideas for such usage:

- The automatic generation of code templates for the Glib Object system.
- The generation of code to transform data structures to external data representations, for instance using XDR.

- The generation of wrappers for remote procedure calls, for instance via XML RPC protocol.

- The generation of various utility functions for data structures – allocation, printing, reading from files, etc.

Some of these tasks can be done using scripting languages or automated program generators such as `autogen` [122]. The advantage of using SWIG for these tasks lies in the fact, that SWIG is fully aware of the C/C++ type system and provides specialised tools for manipulating types, function arguments, class hierarchies, etc.

Of course building and compiling new SWIG modules for some simple or unique tasks could be an overkill. That can be helped however if:

a) SWIG is turned into a function of some scripting language,

b) SWIG is used to generate scripting interface to its own API so new code generation modules can be created in scripting language.

The above modifications to SWIG open a whole new area of SWIG applications.

10.7 Conclusions

This Chapter presented the motivation for the close investigation of the tools for automatic building of the interface modules for the multi-language programming. When building hybrid systems such tools are most of the time treated as the black boxes and users do not need to know their internals. However, as illustrated in this Chapter, a closer scrutiny of such tools might be necessary to enable some special customisations. Investigating the tool's internal working can also lead to some new ideas, such as the one presented in section 10.5 regarding the use of the SWIG interface specification as the scientific interface definition language.

The main case study for this Chapter was how to build the SWIG extension for the Ch scripting language. Section 10.2 gave a detailed account on how to map

C/C++ features to Ch and section 10.3 gave some examples of using Ch SWIG module.

It should be noted that though in this thesis Python is advocated as the scripting language of choice for the scientific computing, it is also worth to consider Ch programming language as an auxiliary language for rapid prototyping. One of the strengths of Ch is that it offers an interpreted C/C++ environment with a rich standard library. Looking from this point of view it would be advantageous to be able to link Ch and Python and the development of a technique for this was presented in section 8.4.

The experimentation with extending SWIG led to the idea of using SWIG interface specification as an universal SIDL. A close inspection of the syntax of SWIG interface specification showed that it has the capacity and flexibility to describe the interface features of most of the languages used in the scientific computing. Moreover, by following this approach one makes sure that the syntactic constructions are backed by the very flexible implementation of the SWIG compiler thus enabling to quickly build the necessary tools.

This Chapter closes the part of the thesis devoted to the development of various aspects of hybrid systems. In appendix A one particular application of a hybrid system is considered, namely the building of web based interface to scientific tools. Providing such interface is the task which requires the consideration of the whole spectrum of problems ranging from an effective number crunching to the quirks of Internet protocols. Hybrid systems, by providing the multiple views on software tools, are the natural environment for such development.

Chapter 11

Conclusions

11.1 Remark

In this thesis practical aspects of the integration of software components for scientific simulations were investigated. Stressing the practical aspect of this research is not without a reason, because it partially deals with problems being within the domains of interest to software engineers, and as such, it cannot be done completely *in isolation from the conditions imposed by reality.* However, by talking about reality we arrive at a subtle point. The goal of most classical engineering *activities* is shaping physical objects or processes, so users can benefit from them or are protected from harmful interaction. Dealing with physical aspects has this enormous advantage that it allows the assessment of the quality of engineering actions and measuring them quite objectively. In the case of software engineering the situation is completely different, as the object of its actions is information, and interaction between humans and information. Of course, the aspect of interaction between human and its environment is present in any engineering activity, however in this case it is completely immaterial, which makes it hard to grasp and quantise. The physical limits to which an environment can change without inducing harm on humans or make them feel uncomfortable has been well recognised, and the rules of optimal shaping of our environment are clearly expressed by ergonomics studies. Going out

of the realm of physical interactions we somehow loose the firm ground on which we can base our decisions and predictions.

This above remark should be kept in mind when making choices while undertaking software development. Computer programs can be judged by very objective and strict criteria of memory utilisation, CPU performance, theoretical time and memory complexity. However, programmers very often talk about much more abstract criteria like expressiveness, clarity, or simple beauty. Taken to a broader context, software solutions should take into account not only the pure technical side of a problem (expressed in programming language statements) but also non-technical aspects like the profile of users, their experience, purpose of using the software and alike. A very bold example of that aspect is a comparison of two software solutions where one has tip-top, extra optimised, well performing code, but without documentation or comments and with a rather clumsy installation procedure, with the other being a not so well performing code, but with good documentation, examples and automatic installation solution. The choice between the two should hardly be a dilemma.

11.2 Summary

The above thought was also a reason why in Chapter 1 the different kinds of users of scientific simulation codes were considered and it was indicated that the computational scientists doing mostly theoretical and educational research were the targeted audience. In that Chapter two general claims and two two specific theses were stated. The two first claims or rather observations are that:

1) In order to reach the next level in developing scientific simulation codes more attention should be paid to component programming and software component integration,

2) It is becoming harder to work in a homogeneous environments, especially single

programming language, ones.

On the basis of the two above statements this dissertation presented two original theses that:

3) In the area of scientific simulations it is possible to overcome difficulties related to the integration of software components by introducing the concepts of the geometry bus and grid bus, as mechanisms for linking software tools based on geometry and grid abstractions. The mechanism proposed is based on the approach of linking APIs instead of integration based on data format.

4) In order to be fully effective geometry and grid buses expressed in system programming languages have to have their counterparts in scripting programming languages.

Chapters 2, 3 and 4 give the support for the theses 1) and 2). Chapter 2 discusses the need for software integration and explains how advances in scientific simulation systems shape their required characteristics in terms of components heterogeneity, communication with distinct components and dynamic adoption of new solutions. It also points to a very important issue of validation and verification of simulation codes, showing how software integration may help with a more common adoption of verification and validation practices.

Chapter 3 is an overview of the most common software integration techniques. While sketching the background of available solutions, it also points to the problem of standardisation failures and standardisation costs, indicating that also in this case practical ways of enabling software interoperability can help in faster convergence to commonly acceptable solutions.

Chapter 4 gives characterisation of modern scripting languages and their use in scientific simulation environments. It introduces the notion of a *hybrid simulation system*. Then it shows possible components from which such hybrid system can be composed, and points to one particular combination of them that seems to be

the most promising. This Chapter also deals with the fallacy of treating scripting languages as inefficient tools unsuitable for numerical simulations, showing that depending on particular situation, gains in system development time might compensate for longer execution times.

Chapter 5 is an elaboration of thesis 3). It introduces the concepts of the geometry bus and the grid bus, points to their role in creating flexible pre- and post-processing tools. It further develops the concept of the buses by introducing the GAGES framework and stressing the importance of its layered structure.

Chapters 6 and 7 are detailed studies of the geometry and grid buses, respectively. They point to the underlying mathematical concepts on which each bus is based. They especially stress the rich mathematical structure and implementation variability. Both Chapters also discuss the sample implementation of the buses, especially pointing to missing components and showing how they can be developed.

Chapter 8 is a practical demonstration of the claim 4). By using the setup of the hybrid system proposed in Chapter 4, this Chapter investigates several case studies of building systems with SWIG Python interfaces to the grid and geometry tools discussed in Chapters 6 and 7.

In Chapter 9 an important development related to the connection between grid bus and geometry bus is presented. This Chapter explains what is the place of surface mesh generation in connecting geometries and grids, points to insufficient saturation with open source surface mesh generators and presents a detailed study on developing one. The development fully utilises the advantages of the GAGES framework and it is a crown argument for supporting theses 3) and 4).

Chapter 10 and 11 are steps toward future extension and utilisation of the effort presented in previous chapters. Chapter 10 describes the development of new software integration tools showing the example of extending SWIG for the Ch language. Beside creating a new interesting tool, the research presented in this Chapter leads to a very interesting and original concept of using SWIG interface files as a universal

scientific interface description language (SIDL).

With respect to the thesis objectives stated in section 1.2 the contribution of this thesis can be summarised as follows:

1. "To design an effective methodology and architecture for connecting components of simulation systems in computational mechanics and especially the ones based on the finite element method." This objective was fully met by the development presented in Chapter 5. That Chapter introduced the GAGES architecture concentrated around two main aspects. The first aspect concerns the data flow from geometry oriented data structures towards grid oriented data structures. This aspect of the GAGES architecture was formalised by the notion of geometry and grid buses. It was stressed in Chapter 9 that an important part of GAGES architecture is the connection between the buses in terms of mesh generation tools. The second aspect is the ability to select between compiled and scripting implementation levels. In the GAGES architecture this is visible by the requirement that grid bus geometry buses are constructed as hybrid systems. Chapter 5 also presents a practical methodology for constructing the respective buses. This methodology uses the bottom-up approach based on selecting an appropriate base library, relies on generic programming techniques, decomposes the architecture into four layers that balance between development and execution efficiency, and stresses the equal importance of compiled and interpreted interfaces. Presentation related to this objective constitutes important and original contribution of this thesis.

2. "To introduce the notion of hybrid system." Some key points of the presented methodology were formalised by the introduction of the concept of hybrid systems in Chapter 4. The notion of a hybrid system stresses the importance of scripting interfaces. Chapter 4 presents several possible combinations of components that can yield a hybrid system, and in particular indicates one of these combinations as the exemplar hybrid system, that in the light of the

presented case studies performs well. A detailed investigation of the interface generation aspect of hybrid systems in Chapter 10 was the impulse for the reflections on a more generic mechanism for specification of component interfaces. This reflection yielded an original idea of using the SWIG specification as an universal Scientific Interface Description language.

3. "To build a concrete framework based on the presented architecture." Such framework has been built and several aspects of its development were presented in Chapter 6 to Chapter 9. The implementation of this framework is based on two libraries: OpenNURBS for geometries and `GrAL` for grids. Reporting on this development provides valuable guide for other designers and programmers.

4. "To show feasibility of the proposed methodology." The feasibility of the proposed methodology was shown by connection of the full range of pre- and postprocessing tools to the framework described in the previous point. The tools were of a very different character – both libraries and programs, and both object oriented and structural codes. In this feasibility study each tool was connected in such a way as to enable access to it on the compiled as well as on the scripting level. This shows that the concept of a hybrid system can be realised and that the presented methodology can be successfully applied in practice. An important contribution of this thesis is to indicate a verified combination of software tools for components connection, as well as presentation of key problems and their possible solutions. This thesis can be used as a road map for the researchers investigating the design and implementation of component based simulation systems.

11.3 Discussion

The case studies presented in this dissertation showed that choosing an approach based on linking APIs instead of developing data format descriptions leads to prac-

tical working solutions. It can be argued that data format descriptions are more fundamental than APIs, as the API represents particular implementation choices. This is of course true, but selecting a very generic API such as GrAL, one mitigates dependencies on a particular implementation decision. Of course nothing comes for free and the price to pay for it, is a higher complexity of grid or geometry bus. Selecting GrAL for a grid bus backbone proved to give an extremely flexible and open solution but also a quite demanding one, from the developers point of view. GrAL uses very advanced template meta-programming techniques which require skills and experience. This is the reason why the in geometry bus, which was developed later, a less generic but conceptually easier solution based on OpenNURBS was used.

One very valuable contribution yielded by working on a GrAL based bus was a series of programming solutions for wrapping in Python a highly template based C++ code such as GrAL. It showed also the quality of SWIG, and that selecting it as a component of the advocated hybrid system was a good decision. The Python interface to GrAL, besides making it easier to use a grid bus, created a very convenient environment for exploring the GrAL concepts too.

The assumption on which the geometry bus and grid bus concepts were built proved to be correct when developing the surface mesh generator presented in Chapter 9. This generator was built when both buses were in an advanced stage and linked already several tools. This enabled the development effort to be concentrated on important algorithmic aspects and the software architecture, leaving low level details to be provided via bus connected tools. For instance, as a result of the visualisation components, it was possible to visualise and graphically follow the generator development from the very beginning.

Important, both from a practical and theoretical point of view, was the development of the SWIG module for the Ch language. On a practical side it enriches the set of languages supported by SWIG and makes it easier to integrate Ch with various numerical libraries. On theoretical side it has lead to development of the concept

of SWIG interface files as a universal Scientific Interface Definition Language. The SWIG interface files proved to be translatable into several concrete languages and this is the main point of any SIDL. Additionally the case study presented in Chapter 10 showed that it is relatively easy to extend SWIG both on code generation side as well as in terms of special SWIG directives. The development of SWIG directives by using the enriching C++ syntax allows the SIDL to be shaped in many ways. However, this is still done in a framework of the SWIG implementation, so at all time one can be sure that the devised SIDL can be translated into a concrete language.

The final conclusion from the above summary and discussion is that the objectives of this thesis were met in full, and the proposed approach to construction of component based simulation systems was verified and practically proved.

11.3.1 Further work

The results obtained while working on this dissertation provide encouragement for further studies. One possible direction of research is to investigate the building of alternative GAGES frameworks based on different libraries for geometry and grid bus backbones. For instance, it would be interesting to build a GAGES framework based on the base libraries implemented in FORTRAN 77 or FORTRAN 90 and using `f2py` as the interface generator between FORTRAN and Python. This would be of practical value because a there is still a lot of legacy code written in FORTRAN 77 and FORTRAN 90 (and subsequent versions), besides C/C++ and is the most popular language for writing the scientific simulation codes. In this case, in some applications, the Python interface could be used as a transport layer for mixing components written in C++ and FORTRAN 90, as a straightforward mixing of code implemented in these languages in not an easy task. This is in contrast to mixing FORTRAN 77 and C, which task, though not standardised and compiler dependent, is quite common. Building the framework based on FORTRAN would strengthen the arguments for feasibility of the proposed methodology.

Another option is to stay within the area of C/C++ based tools but selecting a different base libraries for geometry bus or grid bus. An appealing candidate for the geometry bus backbone is the CGAL library. Alternatively it would be very interesting to base the geometry bus on a different set of underlying concepts other than B-Reps with NURBS description. For example the Djinn project [157] defines a representation independent API for geometry modelling. This is exactly in the spirit of grid bus and it would be interesting to see the merging of Djinn and GAGES architecture.

Remaining at the implementation level it would be of practical value to strengthen the already implemented GAGES framework with links to commercial software packages for finite element calculations. Two promising candidates are ANSYS [160] and Abaqus [161]. Both already use scripting languages as part of their interface – Abacus uses Python for description and control of its calculation process, and ANSYS uses Tcl/Tk and Itcl/Itk for implementation of its GUI.

The other direction of research is the automation of building scripting interfaces. Though SWIG and similar tools take much of the burden of creating wrappers from the developer, building interfaces to such large and complex libraries such as `GrAL` or CGAL still require substantial effort. The next generation of tools could be based not on concrete function and class signatures but on generic concepts which are implemented by them. In this way much of the code repetition can be avoided.

Another possible continuation of this thesis would be the investigation of self describing components and automatic component linking. This idea is in the spirit of blackboard architectures, agent based computing and Grid computing. The components are "simply" endowed in the ability to present information about their interface, and an appropriate environment allows the components to automatically match their interfaces, providing, when necessary, an additional interface layer. In this way the user would only specify that he wants to transfer data from object A to B (where A and B could be data files in various formats or in-memory objects) and

the system will automatically find a route for such transfer, ensuring that there is no information loss, or allowing the user to precisely control what part of the data is transferred. The experience and solutions described in this thesis would be a very good starting point for such research.

Finally a very important issue is dissemination of the presented results in the form of easily installable and configurable software packages. This is quite demanding, because in order for them to be useful a lot of detailed documentation and examples should be written. However, the benefits of dissemination of these results is a good driver for that effort.

References

[1] J.A. Ang, T.G. Trucano, and D.R. Luginbuhl. Confidence in asci scientific simulations. HICSS-32, Software Technolog Track, MiniTrack-8 Parallel and Distrubuted Simulations, 2000.

[2] M.C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In R. Cripps, editor, *Proc. 8th IMA Conf. on The Mathematics of Surfaces*, pages 1–20, 1998.

[3] Pipeline & XT. UGS homepage. `http://www.ugs.com/products/open/parasolid/pipeline.shtml`.

[4] M.W. Beall and M.S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40:1573–1596, 1997.

[5] B.H.V. Topping, J. Muylle, P. Iványi, R. Putanowicz, and B. Cheng. *Finite Element Mesh Generation*. Saxe-Coburg Publications, Edinburgh, 2004.

[6] L. Hatton. The T experiments: Errors in scientific software. *IEEE Computatinal Science & Engineering*, 4(2):27–48, 1997.

[7] Simplified Wrapper and Interface Generator. `http://www.swig.org`.

[8] D. Wheeler, J.E. Guyer, and J.A. Warren. FiPy. a finite volume PDE solver using Python. National Insititute of Standards and Technology, June 2005. version 1.0a2 edition.

[9] ASP data specifications 2004-jan.

`http://www.asp.cornell.edu/itr-asp/specs/latest`.

[10] G. Allen, E. Seidel, and J. Shalf. Scientific computing on the Grid. *Byte*, Spring, 2002.

[11] G.A. Articolo. *Partial Differential Equations and Boundary Value Problems with Maple V.* Accademic Press, 1998.

[12] A.I. Beltzer. *Engineering Analysis with MAPLE/MATHEMATICA*. Academic Press, 1995.

[13] G. Berti. *Generic Software Components for Scientific Computing*. PhD thesis, Brandenburgischen Technischen Universität Cottbus, 2000.

[14] J.-D.Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1998.

[15] D. Booth. Integrating python, c and c++. Web page.

[16] H. Borouchaki, F. Hecht, and P.J. Frey. Mesh gradation control. *Int. J. Numer. Meth. Engng*, 43:1143–1165, 1998.

[17] P. Capolsini. Macroc: C code generation within maple. Report no. 151, INRIA-I3S, February 1993.

[18] L.P. Chew, S. Vavasis, S. Gopalsamy, T. Yi Yu, and B. Soni. A concise representation of geometry suitable for mesh generation. In *Proceedings, 11th International Meshing Roundtable*, pages 275–284, 2002.

[19] A. Cleary, S. Kohn, S.G. Smith, and B. Smolinski. Language interoperability mechanisms for high-performance scientific applications. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, October 21-23 1998. LLNL report UCRL-JC-131823.

[20] B. Cockburn, C. Johnson, C.-W. Shu, and E. Tadmor. *Advanced Numerical Approximation of Nonlinear Hyperbolic Equations*, volume 1697 of *Lecture Notes in Mathematics*. pub-spring, 1998. Lectures given at the 2nd Session of the Centro Internazionale Matematico Estivo (C.I.M.E.) held in Cetaro, Italy, June 23-28, 1997.

[21] B. Cockburn. An introduction to the discontinuous galerkin method for convection-dominated problems. In A. Quateroni, editor, *Advanced Numerical Approximation of Nonlinear Hyperbolic Equations*, volume 1697 of *Lecture Notes in Mathematics*, pages 151–268. pub-spring, 1998. Lectures given at the 2nd Session of the Centro Internazionale Matematico Estivo (C.I.M.E.) held in Cetaro, Italy, June 23-28, 1997.

[22] A.L. Cunha, S.A. Canann, and S. Saigal. Automatic boundary sizing for 2d and 3d meshes. In *Proc. Symp. Trends in Unstructured Mesh Generation*, pages 65–72. Amer. Soc. Mechanical Engineers, Jun 1997.

[23] T. Dahlgren, T. Epperly, and G. Kumfert. Babel users' guide. CACS, Lawrence Livermore National Laboratory, January 2004. version 0.9.0 edition.

[24] J.W. Eaton. Octave: Past, present, and future. In *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, 2001.

[25] J.W. Eaton. Octave: A high-level interactive language for numerical computations, February 1992.

[26] J.W. Eaton and J.B. Rawlings. Ten years of Octave - recent developments and plans for the future.

[27] J. Ebert. A versatile data structure for edge-oriented graph. *Communications of the ACM*, 30(6):513–519, jun 1987.

[28] Tom Epperly, Scott R. Kohn, and Gary Kumfert. Component technology for high-performance scientific simulation software. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 69–86, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[29] L.H. de Figueiredo. *Graphics Gems V*, chapter Adaptive Sampling of Parametric Curves, pages 173–178. Academic Press, Inc., 1995.

[30] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid. *International Journal of Supercomputer Applications*, 2001.

[31] P.J. Frey. Anisotropic metrics for mesh adaptation. ECCOMAS, 24–28 July 2004.

[32] P.J. Frey and H. Borouchaki. Surface mesh quality evaluation. *Int. J. Numer. Meth. Engng*, 45:101–118, 1999.

[33] D. Funaro. *Polynomial Approximation of Differential Equations*, volume m8 of *Lecture Notes in Physics*. Springer-Verlag, 1992.

[34] D. Funaro. *Spectral Elements for Transport-Dominated Equations*, volume 1 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, 1997.

[35] E. Gallopoulos, E. Houstis, and J.R. Rice. Problem-solving environments for computational science. *IEEE Computational Science & Engineering*, 1:11–23, 1994.

[36] R. Glowinski, E.Y. Rodin, and O.C. Zienkiewicz, editors. *Energy Methods in Finite Element Analysis*. pub-wiley, 1979.

[37] C. Gomez. Macrofort: a fortran code generator in maple. Report no. 119, INRIA, May 1990.

[38] C. Gomez and T. Scott. Maple programs for generating efficient fortran code for serial and vectorised machines. *Computer Physics Communications*, 115:548–562, 1998.

[39] J.L. Gross and T.W. Tucker. *Topological Graph Theory.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1987.

[40] L. Hatton and A. Roberts. How accurate is scientific software. *IEEE Transactions on Software Engineering*, 20(10):785–797, 1994.

[41] M. Henle. *A Combinatorial Introduction to Topology.* A Series of Books in Mathematical Science. W. H. Freeman and Company, 1979.

[42] W.T. Hewitt and D. Yip. The nurbs procedure library. Technical Report CGU 76, Manchester Computing Centre, Computer Graphics Unit, 1992.

[43] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[44] E.N. Houstis and J.R. Rice. On the future of problem solving environments, March 2000.

[45] E.N. Houstis. The role of problem solving environments in engineering and mathematics education. In *6th International Conference on Technology in Mathematics Teaching. Volos, Greece,*, 2003.

[46] B. Howe, D. Maier, and A. Baptista. A language for spatial data manipulation. *Journal of Environmental Informatics*, 2003.

[47] C.L. Kinsey. *Topology of Surfaces.* Undergraduate Texts in Mathematics. Springer-Verlag, 1993.

[48] H. Koike. Evolution of cfd software form academic code to practicl engineering software. *Journal of Wind Engineering and Industrial Areodynamics*, 81:41–55, 1999.

[49] C.K. Lee. A new finite point generation scheme using metric specification. *Int. J. Numer. Meth. Engng*, 48:1423–1444, 2000.

[50] C.K. Lee. On curvature element-size control in metric surface mesh generation. *Int. J. Numer. Meth. Engng*, 50:787–807, 2001.

[51] T. Lindgren, J. Sanchez, and J. Hall. *Graphics Gems IV*, chapter Curve tesselation criteria through sampling, pages 262–265. Academic Press, Inc., 1992.

[52] R.I. Mackie. Extensibility of finite element class systems – a case study. *Computer and Structures*, 82:2241–2249, 2004.

[53] J.G. Michopoulos. Development of the finite element modeling markup language. In *Proceedings of DETC'02*, 2002.

[54] J. Michopoulos, P. Mast, T. Chwastyk, L. Gause, and R. Badaliance. FemML for data exchange between FEA codes. In *"ANSYS Users' Group Conference*, University of Maryland COllege Park MD, October 2001.

[55] G.F. Moita and M.C. Pinheiro. Towards a standard for finite element data exchange using XML. In *CD-Rom Conference Proceedings of the 7th U.S. National Congress on Computational Mechanics*, pages 1–2, Albuquerque : USACM, 2003.

[56] M.B. Monagan, K.O. Geddes, K.M. Heal, G. Labhan, S. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 7 Programming Guide*. Waterloo Maple Inc., 2001.

[57] M.B. Monagan, K.O. Geddes, K.M. Heal, G. Labhan, and S. Vorkoetter. *Maple V Programming Guide*. Springer, 1996.

[58] A.K. Noor. Computational structures technology: leap frogging into twenty-first century. *Computers and Structures*, 73:1–31, 1999.

[59] A.K. Noor. New computing systems and future high-performance computing environment and their impact on structural analysis and design. *Computer & Structures*, 64(1-4):1–30, 1997.

[60] J.T. Oden and L.F. Demkowicz. *Applied Functional Analysis.* CRC Series in Computational Mechanics and Applied Analysis. CRC, 1996.

[61] J.T. Oden and J.N. Reddy. *Variational Methods in Theoretical Mechanics.* pub-springer, 1976.

[62] S.J. Owen and S. Saigal. Surface mesh sizing control. *Int. J. Numer. Meth. Engng*, 47:497–511, 2000.

[63] P. Parkinson and P. Shulman. Putting the pieces together – the promise of mixed language programming. *Dedicated Systems Magazine*, Q1, 2005.

[64] J. Peng, D. Liu, and K.H. Law. An engineering data access system for a finite element program. *Advances in Engineering Software*, 34:163–181, 2003.

[65] J. Peng and K.H. Law. Building finite element analysis program in distributed services environment. *Computer and Structures*, 82:1813–1833, 2004.

[66] J.R. Rice. A perspective on computational science in the 21st century. *Computing in Science and Engineering*, 1(2), 1999.

[67] J.R. Rice and R.F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Computational Science & Engineering*, 3(Fall):44–53, 1996.

[68] J.R. Rice. Future scientific software systems. *Computational Science and Engineering*, 4(2):44–48, 1997.

[69] E. Houstis, E. Gallopoulos, R. Bramley, and J.R. Rice. Problem-solving environments for computational science. *IEEE Computational Science & Engineering*, 4(3):18–21, 1997.

[70] J. Riehl. Pyfront: Conversion of python to c extension modules. Web page, 1998.

[71] J.A. Sanders and J.P Wang. Combining maple and form to decide on integrability questions. *Computer Physics Communications*, 115:447–459, 1998.

[72] J. Sang and C. Kim. Developing corba-based distributed scientific applications from legacy fortran programs. Technical Report E-12204, National Aeronautics and Space Administration John H. Glenn Research Center at Lewis Field, July 2000. `http://gltrs.grc.nasa.gov/cgi-bin/GLTRS/browse.pl?2000/TM-2000-209950.html`.

[73] S. Sechrest. Tutorial examples of interprocess communication in berkeley unix 4.2 bsd. Technical Report UCB/CSD-84-191, EECS Department, University of California, Berkeley, 1984.

[74] M.S. Schephard. Automatic generation of finite element models. In M. Papadrakakis, editor, *Solving Large-scale Problems in Mechanics*, chapter 13. John Wiley & Sons Ltd, 1993.

[75] I.M. Singer and J.A. Thorpe. *Lecture Notes on Elementary Topology and Geometry.* Springer-Verlag, 1976.

[76] C.L. Spiel. Da coda al fine: Pushing ocatave's limits, 2000.

[77] A. Stahel. A fem algorithm in octave, August 2002. version 0.1.4.

[78] E. de Struler, J. Hoeflinger, L.V. Kale, and M. Bhandarkar. A New Approach to Software Integration Frameworks for Multi-physics Simulation Codes. In *Proceedings of IFIP TC2/WG2.5 Working Conference on Architecture of Scientific Software, Ottawa, Canada*, pages 87–104, October 2000.

[79] G.K. Thiuruvathukal, J.P. Shafaee, and T.W. Christopher. *Web Programming. Techniques for Integrating Python, Linux, Apache, and MySQL.* Prentice Hall PTR, 2002.

[80] I. Tuomi. The lives and death of Moorse's law. *First Monday*, 7(11), 2002.

[81] M. Vigo and N. Pla. *Computer & Graphics*, 24(2), 2000.

[82] K.-P. Vo. The discipline and method architecture for reusable libraries. *Software – Practice and Experience*, 30:107–128, 2000.

[83] A.T. White. *Graphs, Groups and Surfaces*, volume 8 of *North-Holland Mathematics Studies.* North-Holland, 1984.

[84] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG.* Prentice Hall, 2nd edition, 1996.

[85] J.T. Meindl. Beyond moore's law: The interconnect era. *IEEE Computing in Science and Engineering*, January/February 2003.

[86] S. Hamilton. Intel research expands Moore's law. *IEEE Computer*, January 2003.

[87] Z. Turk. Internet information and communication systems for civil engineering: A review. In B.H.V. Topping, editor, *Civil and Structural Engineering Computing: 2001*, pages 1–26. Saxe-Coburg Publications, 2001.

[88] B.H.V. Topping, editor. *Computational Mechanics for the Twenty-First Century.* Saxe-Coburg Publications, Edinburgh, 2000.

[89] M. Cross, C. Baily, K. Pericleous, T.N. Croft, and G. Taylor. Issues in the design of computational software technology for the simulation of closely coupled multi-physics processes on prallel systems. In B.H.V. Topping, editor, *Computational Mechanics for the Twenty-First Century*, pages 205–217. Saxe-Coburg Publications, Edinburgh, 2000.

[90] E.S. Raymond. *The Art of UNIX Programming.* Prentice Hall PTR, 2004.

[91] S. Loosemore. *The GNU C Library: System & Network Applications.* GNU Press, 2004. For GNU C Libraries version 2.3.x.

[92] SoftIntegration. *The Ch Language Environment. SDK User's Guide*, version 5.1 edition, 2005.

[93] SoftIntegration. *The Ch Language Environment – User's Guide*, version 5.0 edition, 2005.

[94] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, 1998.

[95] W.J. Gordon and L.C. Thiel. Transfinite mappings and their application to grid generation. In J.F. Thompson, editor, *Numerical Grid Generation.* Elsevier, 1982.

[96] M. Lacage. The Glib Object system v0.10.0, 2004. `http://le-hacker.org/papers/gobject`.

[97] E. Arge, A.M. Bruaset, and H.P. Langtangen, editors. *Modern Software Tools in Scientific Computing.* Birkhäuser Press, 1997.

[98] G. Karypis and V. Kumar. *Metis. A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computiong Fill-Reducing Orderings of Sparse Matrices*, version 4.0 edition, 1998.

[99] M. Lee and A.A. Stepanov. The standard template library. Technical report, Hewlett-Packard Laboratories, February 1995.

[100] J. Siek and A. Lumsdaine. The matrix template library. MTL Home Page, 1999.

[101] D.R. Musser and A.A. Stepanow. Generic programming. In *Symbolic and algebraic computation: International Symposium ISSAC'88, Rome Italy, July 4–8, 1988:proceedings*, number 358 in LNCS, pages 13–25. Springer, 1989.

[102] D.R. Musser and A.A. Stepanov. Algorithm-oriented generic libraries. *Software Practice and Experience*, 24:623–642, 1994.

[103] The cgal project. The CGAL home page – Computational Geometry Algorithms Library, 1999.

[104] SGI. *SGO Standard Template Library Programmer's Guid*, since 1996.

[105] J.R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In M.C. Lin and D. Manocha, editors, *Applicd Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1996. From the First ACM Workshop on Applied Computational Geometry.

[106] J.R. Shewchuk. Triangle: A two-dimensional quality mesh generator, 2004.

[107] C. Decusatis. Grid computing: The next (really, really) big thing. *Byte*, pages 6–13, Spring 2002.

[108] D.E. Stevenson. A critical look at quality in large-scale simulaitons. *Computing in Science & Engineering*, pages 53–63, May-June 1999.

[109] C.F. Ollivier-Gooch. *GRUMMP Version 0.2.1 User's Guide*. University of British Columbia, Department of Mechanical Engineering, September 2002.

[110] C. Boivin and C.F. Ollivier-Gooch. Guaranteed-quality triangular mesh generation for domains with curved boundaries. *International Journal of Numerical Methods in Engineering*, 55(10):1185–1213, 2002.

[111] C.F. Ollivier-Gooch and C. Boivin. Guaranteed quality simplicial mesh generation with cell size and grading control. *Engineering with Computers*, 17(3):269-286, 2001.

[112] JOSTLE – graph partitioning software. `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.

[113] E. Boender, W.F. Bronsvoort, and F.H. Post. Finite-element mesh generation from constructive-solid-geometry models. *Computer-Aided Design*, 26(5):379-391, May 1994.

[114] J.J. Shah J.D. Summers, Z. Lacroix. Collaborative mechanical engineering design – representation and decision issues. `http://citeseer.ist.psu.edu/708473.html`.

[115] Electronics Data Systems Corporation. *Parasolid XT Format*, May 2003.

[116] H. Spencer. How to steal code -or- inventing the wheel only once. In *USENIX Winter*, pages 335–345, 1988.

[117] SETI home. `http://setiathome.berkeley.edu`.

[118] Scilab Home Page. `http://www.scilab.org`.

[119] Swig-1.3 development documentation, 2005. `http://www.swig.org/Doc1.3/index.html`.

[120] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit. An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., 3rd edition, 2003.

[121] Bison. `http://www.gnu.org/software/bison`.

[122] The AutoGn home page. `http://www.gnu.org/software/autogen`.

[123] The Curl home page. `http://curl.haxx.se`.

[124] GTS – The GNU Triangulated Surface Library.
http://gts.sourceforge.net.

[125] Catalog of OMG Specifications. http://www.omg.org/technology/documents/spec_catalog.htm.

[126] HDF5 Mesh API. http://hdf.ncsa.uiuc.edu/HDF5/papers/prototypes/mesh/.

[127] TetGen – A Quality Tetrahedral Mesh Generator. http://tetgen.berlios.de.

[128] NETGEN. http://www.hpfem.jku.at/netgen/.

[129] Open Visualization Data Explorer. http://www.opendx.org/.

[130] VTK Home Page. http://www.vtk.org.

[131] OpenNURBS Home Page. http://www.opennurbs.com.

[132] I. Foster. What is the Grid? A three point checklist. *Grid Today*, 1(6), July 2002. http://www.gridtoday.com/02/0722/100136.html.

[133] Richard Hamming – Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Richard_Hamming.

[134] The plotutils package. http://www.gnu.org/software/plotutils.

[135] GrAL - Grid Algorithms Library Home Page.
http://www.math.tu-cottbus.de/~berti/gral.

[136] The MayaVi data visualizer. http://mayavi.sourceforge.net.

[137] Chaco: Software for Partitioning Graphs.
http://www.cs.sandia.gov/~bahendr/chaco.html.

[138] ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[139] Eli: An Integrated Toolset for Compiler Construction. http://eli-project.sourceforge.net.

[140] ANTLR parser generator. http://www.antlr.org/.

[141] Fast Light Toolkit FLTK. http://www.fltk.org.

[142] S.C. North. *Agraph tutorial.* AT&T Shannon Laboratory, Florham Park, NJ, USA, July 2002. www.graphviz.org/Documentation/Agraph.pdf.

[143] GraphViz. http://www.graphviz.org.

[144] PLplot Pome Page. http://plplot.sourceforge.net.

[145] Gnuplot interfaces in ANSI C. http://ndevilla.free.fr/gnuplot.

[146] Gnuplot home page. http://www.gnuplot.info.

[147] Project CHASE. http://www.15.pk.edu.pl/~putanowr/chase/chase_index.html.

[148] Cdt container dictionary type package. http://www.research.att.com/~gsf/download/ref/cdt/cdt.html.

[149] SOAP W3C Recommendation. http://www.w3.org/TR/2003/REC-soap12-part0-20030624.

[150] RFC 1014 - XDR: External Data Representation standard. http://www.faqs.org/rfcs/rfc1014.html.

[151] The Netlib. http://www.netlib.org.

[152] A. Mabogunje and L Leifer. 210-np: Measuring the mechanical engineering design process. In *Twenty-Sixth Annual Frontiers in Education Conference on Technology-Based Re-Engineering Engineering Education*, Salt Lake City, UT, 1996.

[153] R. Biswas and R. Strawn. A new procedure for dynamic adoption of three-dimensional unstructured grids. *AIAA-93-0672*, Presented at the 31st Aerospace Sciences Meeting & Exhibit, Jan. 11-14 1993, Reno, NV.

[154] S.D. Connel and D.G. Holmes. 3-dimensional unstructured adaptive multigrid scheme for the Euler equations. *AIAA Journal*, 32:1626–1632, 1994.

[155] Y. Kallinderis and P. Vijayan. Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes. *AIAA Journal*, 31:1440–1447, 1993.

[156] Zpl. `http://www.cs.washington.edu/research/zpl/home/index.html`, December 2006.

[157] Cecil Armstrong and et. al. *Djinn. A Geometric Interface for Solid Modelling.* Geometric Modelling Society, 2000.

[158] H. Adeli and G. Yu. An integrated computing environment for solution of complex engineering problems using the object-oriented programming paradigm and a blackboard architecture. *Computers & Structures*, 54(2):255–265, February 1994.

[159] H. Penny Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *The AI Magazine*, pages 38–53, Summer 1686.

[160] Ansys. `http://www.ansys.com/`, 2007.

[161] Abaqus. `http://www.simulia.com`, 2007.

Appendix A

Scientific applications as web services

A.1 The role of web services in scientific environments

In principle scientific web services do not differ from other web services – they are programs that can be operated remotely over the Internet with the help of an appropriate protocol to deliver the service.

However, the adjective "scientific", besides suggesting the service scope, also indicates that the services may differ from the other ones by having much higher requirements in terms of data volume sent between client and server, higher server utilisation in terms of CPU load, and a smaller number of connections and frequency of use. In addition, scientific web services may need to take into account specific data structures (vectors, matrices, grids) used in scientific simulations.

On the other hand, the motivations for considering the use of web services for scientific work are exactly the same as in other application areas. They are:

- enabling access to specific resources – both hardware and software,
- enabling work in geographically and/or institutionally spread teams,

- mitigating problems related to use of components based on different operating systems and programming languages,
- cutting down development costs.

The use of scientific web services is not restricted only to big, sophisticated projects with large budgets to support complex information processing infrastructure and expensive computer science experts. Quite on contrary, single researchers and small teams can benefit much as being providers and users of simple web services.

It has happened many times, that the author was asked by his friends to do some calculations for them. The author's programs were either his specific modifications and for various reasons inaccessible to others, or the people who asked did not have time to play with installation of these programs, as in some cases it would require switching to another operating system. The role of author was reduced to receiving data, running a program and sending data back. Several times after repeating the above "just one more time, please", the author had a desire to automate the whole procedure, but the costs of automation plus the hope that this is really the last time usually outweighed the obvious gains. In retrospection however, doing the automation at the very first time would be more beneficial. A similar situation was when the author was writing some piece of code for others. Here problems were arising because some utility libraries could not be used, or on targeted systems they were installed in different version, or when the use of different programming languages came into question. Also, when the code needed to be modified and fixed then it was tedious to send and retrieve the code and recompile programs.

In both situations described above the author and his friends would be better off, if they had used web services. In the first case a web service would be a wrapper over a complete program and solution based on a CGI could be used. In the second case a solution based on some kind of remote procedure call could be utilised. For the author it would mean less work and the other party would be happier having done the work faster and being able to use the service at any time.

A.2 Automation of web services building

The advantages of web services are obvious. However the problem with them, as with any novel solution, is that the immediate costs of their application shadow their long term benefits. The costs are on both sides, service providers and service users. In the case discussed above the author would have to implement the service and the other party would have to obtain or implement, an appropriate software client to use the service. The problem is that creating web services is more in the domain of computer scientists than computational scientists. It is hard to expect that someone working on finite element programming and fighting with implementation of another "discontinuous Galerkin method based on natural neighbour finite elements"[1] will have time to additionally investigate the quirks of writing CORBA clients or mundane details of SOAP protocol. Here is the place where automatic interface building and ideas presented in this thesis will have their application.

In the light of the above we may say that web services are another technique for software systems integration. We may talk about coarse grained integration where web service is base on an application as a whole, or about fine grained integration where web services deliver software building blocks like functions or objects.

While there are many possible solution for web services: CORBA, COM, web server extension modules, dedicated servers, SOAP, universal frameworks such as Zope, here two lightweight solutions will be discussed – CGI and XML-RPC, because of relatively simple way their application can be automated.

A.2.1 Coarse grained software integration based on CGI

Though simple and old, CGI still proves to be the most popular solution for delivery of services over Internet. On server side CGI programs can be written in any language but traditionally it has been the domain of scripting languages, particularly Perl, Python, Tcl, Ruby. In their standard distributions these languages offer the

[1]This subject is completely made up, though possible.

most complete set of tools for tasks appearing when processing CGI requests. By extending these languages with the scientific tools as shown in Chapter 8, one obtains very natural environments for building scientific web services. The appreciation for such environments is visible for instance in the Ch language, for which its vendor SoftIntegration, prepared `chcgi` and `chcurl` packages. Also, one can find several examples of web services on SoftIntegration site `http://www.softintegration.com`.

The most common assumption is that for producing CGI request users will use a web browser. Users are presented an HTML based form which they complete with the necessary information. Then, after optional validation on the client side, the form data is sent to the CGI server using HTTP protocol. This solution of course requires user interaction to complete the form. However, the protocol is open and simple, so an appropriate custom CGI client can be written with little effort. This is important, because besides the manual operation on the service input and output data, we usually want to incorporate a web service in a bigger data processing tool chain.

Though not excruciatingly difficult, writing CGI based services can be made even simpler – especially on the client side – with the idea presented in section 10.5. In many situations the client side of the CGI service can be described in terms of a C function call with C data structures describing the form fields. Taking such a description, one can generate an appropriate client automatically. The client might be a stand alone script or a C function for incorporation in a larger program. What such generation process must ensure is the proper translation of input data to the form, which can be sent via HTTP protocol, for instance saving the data in a file and using a POST CGI request. Based on the work presented in Chapter 10, it is possible to modify the SWIG, and with help of such libraries such as **curl** [123], to build respective the CGI client generator.

A.2.2 Fine grained software interaction based on an XML-RPC

RPC as a way for software integration was described in section 3.3. For a long time the use of RPC was hampered by a floating standard of its data structures, and when SUN company came forward with XDR standard [150] it haven't really caught on. Only when bundled with XML standard for data description, XML-RPC has became a popular solution for automating communication. Contrary to a more powerful solutions, XML-RPC is small, consistent and works. XML-RPC could be thought as a partial implementation of SOAP (Simple Object Access Protocol)[149]. SOAP promises to solve many problems which cannot be handled by XML-RPC but though it has some partial implementations they are not fully compatible. Other popular and powerful frameworks for automating communications, JavaBeans and Microsoft's COM do not really stand for cross-language or cross-platform requirements. However, the biggest problem with them is that they are complex, much to complex for the situation described in section A.1 of two researches wanting to exchange some simple routine. An XML-RPC, on the other hand, perfectly matches such a situation.

An advantage of using an XML-RPC for building web services is that it allows relatively simple automation of building both, client and server ends. The XML-RPC protocol is simple, but according to the policy consistently presented in this thesis this protocol will not be relied upon but the API of `xmlrpc-c` library will be used, which helps in building XML-RPC applications. The SWIG module for dealing with such automation would then translate between original C functions calls and `xmlrpc-c` routines. Again, as is the case with CGI based solutions, the most work will be required to properly translate user data types into types supported by the XML-RPC protocol.

A.3 Examples

The biggest problem in creating web services is of course the implementation of the service's server side. To study it, and to show how the tools presented in this dissertation may help with building web services for grids processing, three simple examples were considered.

In principle, web services for coarse grained software integration could be built only on the basis of the CGI support provided by a standard library of a given language, for instance by the Python's `cgi` module. Unfortunately this support suffices only in simple cases. If the service is for doing some serious work, then surely it will grow and become more complex in time. Besides the basic functionality, the real service must deal with such issues as user authentication, activity logging, debugging, security, data persistence and personalisation of services. Adding subsequent features moves the service from a simple CGI application towards an entire web portal. In such a case, tasks like opening database connections, or validating and authenticating sessions start to dominate the development. By doing it only with basic CGI libraries one can quickly obtain unmanageable code or spend much time on designing already implemented solutions.

If one envisages further evolution of simple CGI scripts then it is probably appropriate to consider the use of one of several development frameworks. They simply permit factoring out repeated tasks, and provide a centralised access to common facilities. Examples of such frameworks include:

PHP is a server side scripting language used to develop dynamic server pages. PHP is quite popular as a result of a standard library covering most of the common tasks. The biggest disadvantage of PHP is not a clear separation between PHP scripts and the HTML code. Mixing presentations and the business logic of a web application makes it difficult to modify and extend.

ASP (Active Server Pages) is the Microsoft-based sever side scripting language.

ASP allows the selection of the primary scripting language for implementing business logic. Available are: VisualBasic, Perl, and Python. The biggest problem with ASP is portability. Currently the full ASP API is supported only by Windows IIS servers.

Zope is a Python based framework for web applications, that originally started as a solution to dynamic content management. Zope is implemented in Python with performance-critical pieces written in C. Zope consists of two main components: Z Object Data Base (ZODB) and Z server, which is a persistent, stand alone, object, Web and FTP server. The main disadvantage of Zope is its apparent complexity.

These three choices were investigated for the purpose of this dissertation. The PHP solution was rejected on the ground of the necessity to learn another language and the unclear separation of presentation and business logic. An ASP solution, though attractive, could not be used in author's UNIX/Linux dominated environment. Finally Zope, though quite appealing because of its Python roots was rejected because it appeared too complex at first sight. Other possible solutions which were not closely investigated are: ColdFusion, JavaServlets and JSP.

What is required is a moderate complexity solution, open source, Python based, with clear separation of presentation and business logic aspects. Such a solution was found in the form of the Slither web development framework [79]. In addition to the mentioned features, Slither is quite well described in book [79]. Sadly, at the time of writing this thesis, the Slither project does not seem to be actively maintained.

A.3.1 METIS partitioner interface

This application allows the generation and the view of partitioning of 2D mixed elements meshes. The input consists of a mesh file in a Complex2D format from the GrAL library, the number of partitions and nodes numbering offset (0 or 1). On

output a PNG image of a partitioned mesh is presented together with a link to a file containing the partition data.

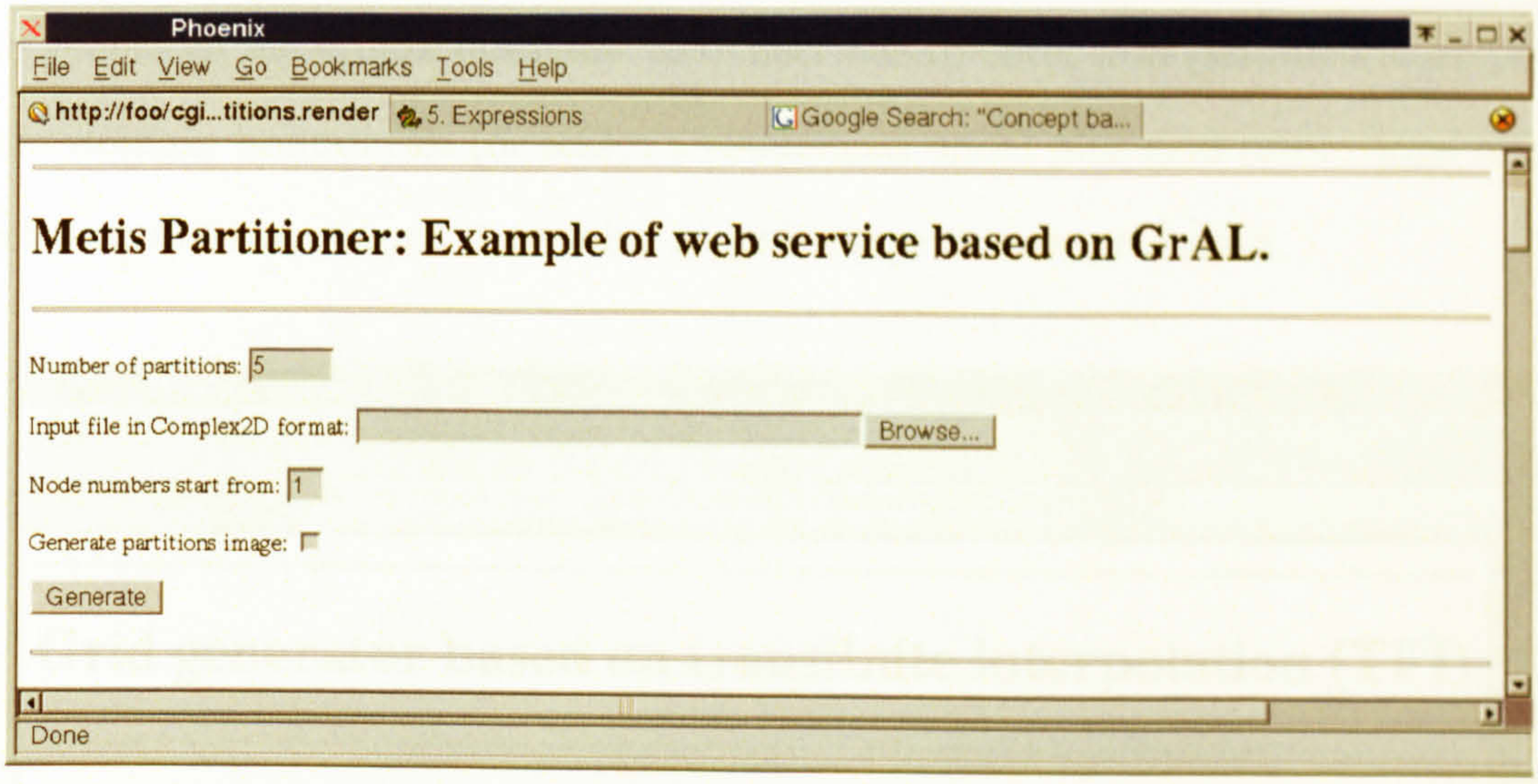


Figure A.1: Data input page for mesh partitioning service based on Metis.

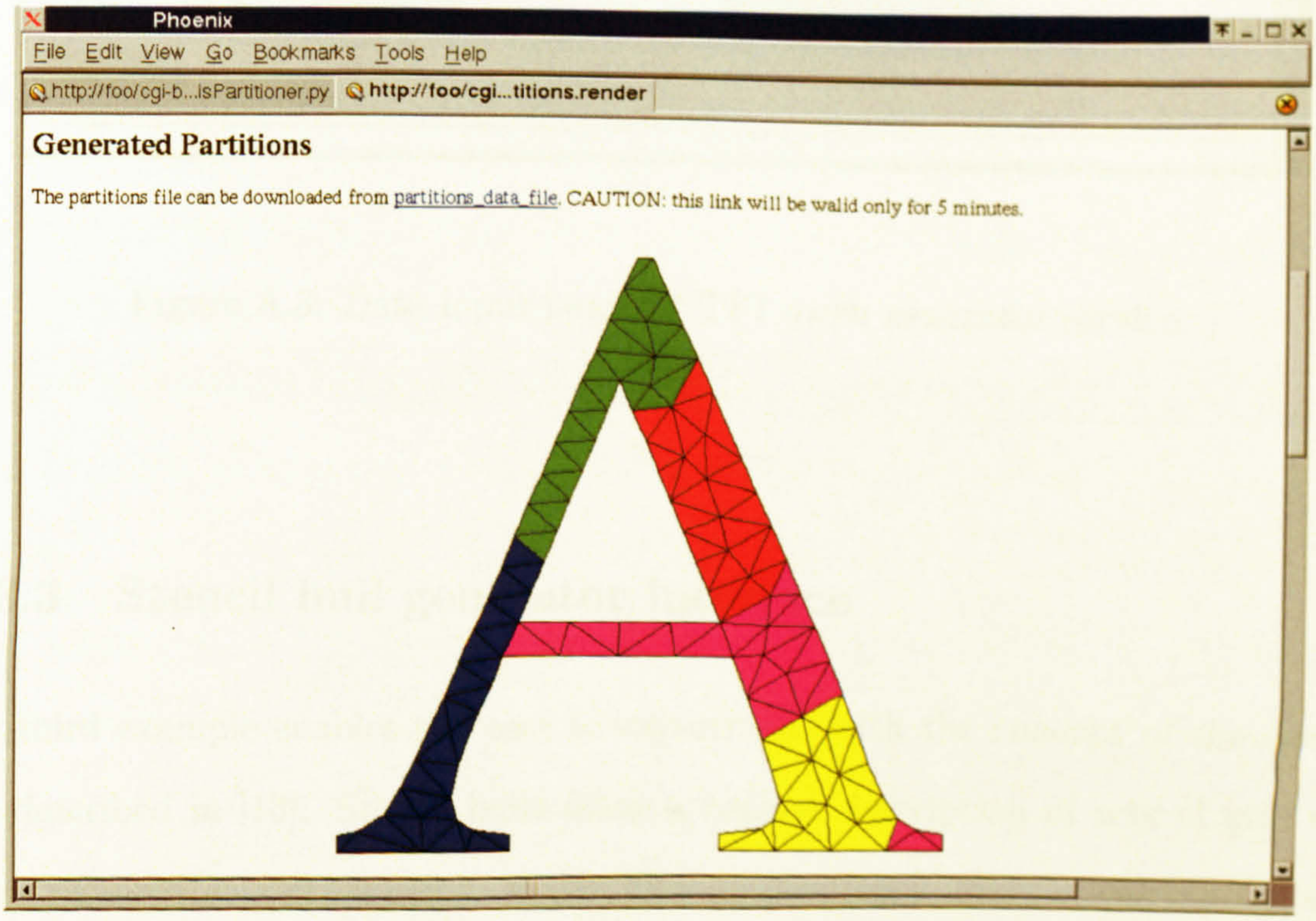


Figure A.2: Results page for the mesh partitioning service.

A.3.2 Interface to a mesh generator based on TFI mapping

The second example permits the user to experiment with a transfinite interpolation mesh generator for square domains. The user has to enter four parametric mappings for the domain boundaries (or select a predefined ones) plus grid resolutions in both directions. On the output it obtains a grid image or raw grid data.

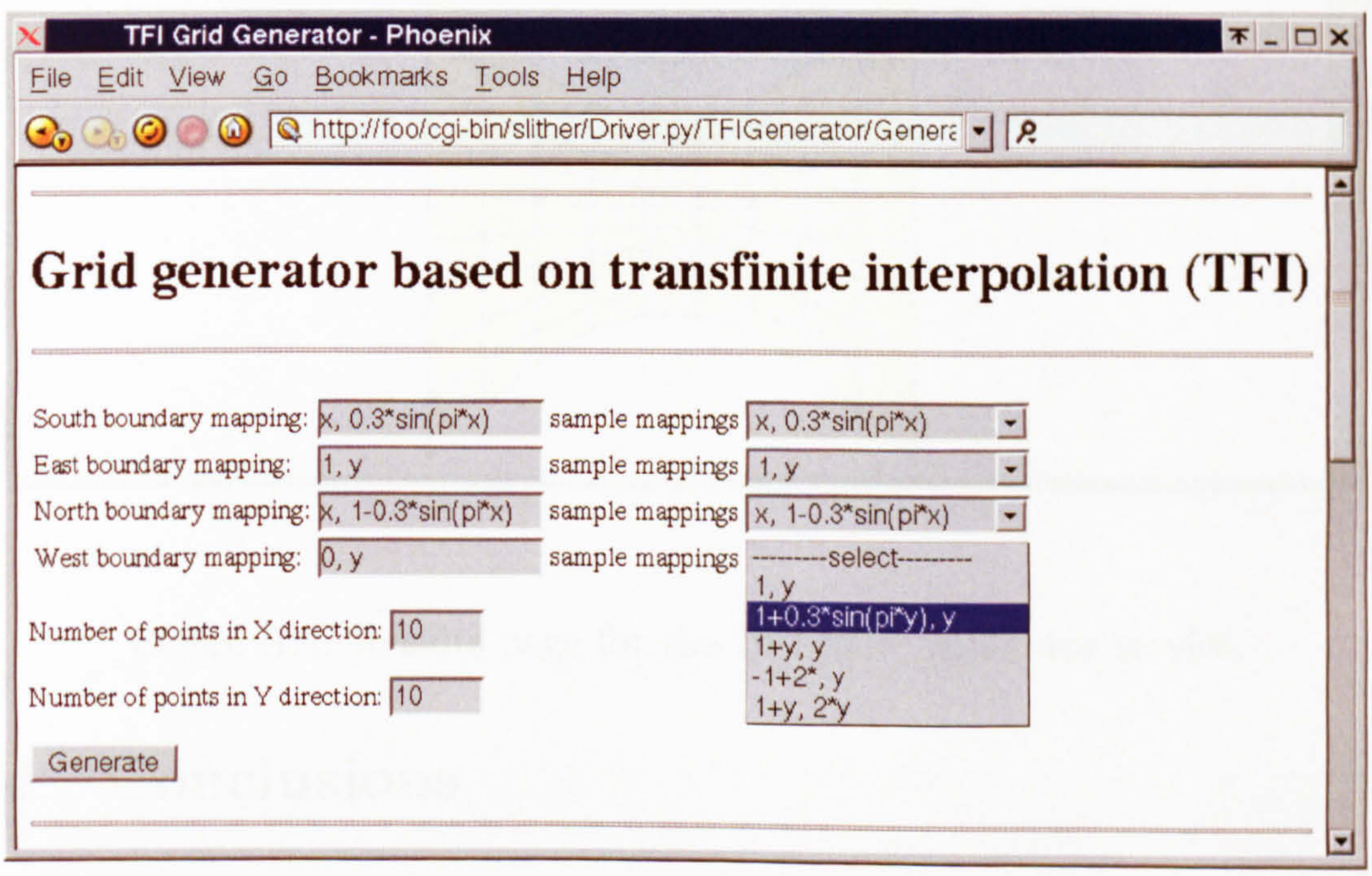


Figure A.3: Data input page for TFI mesh generator service.

A.3.3 Stencil hull generator interface

The third example enables the user to experiment with the concept of the stencil hull described in [13]. Stencil hulls allow a concise description of sets of grid elements required at a given cell or node. This application allows the calculation and visualisation of stencil hulls on a 10 by 10 structured gird. The result of this application is the visualisation of a stencil hull with a vertex and *cell* layers coloured with different colours.

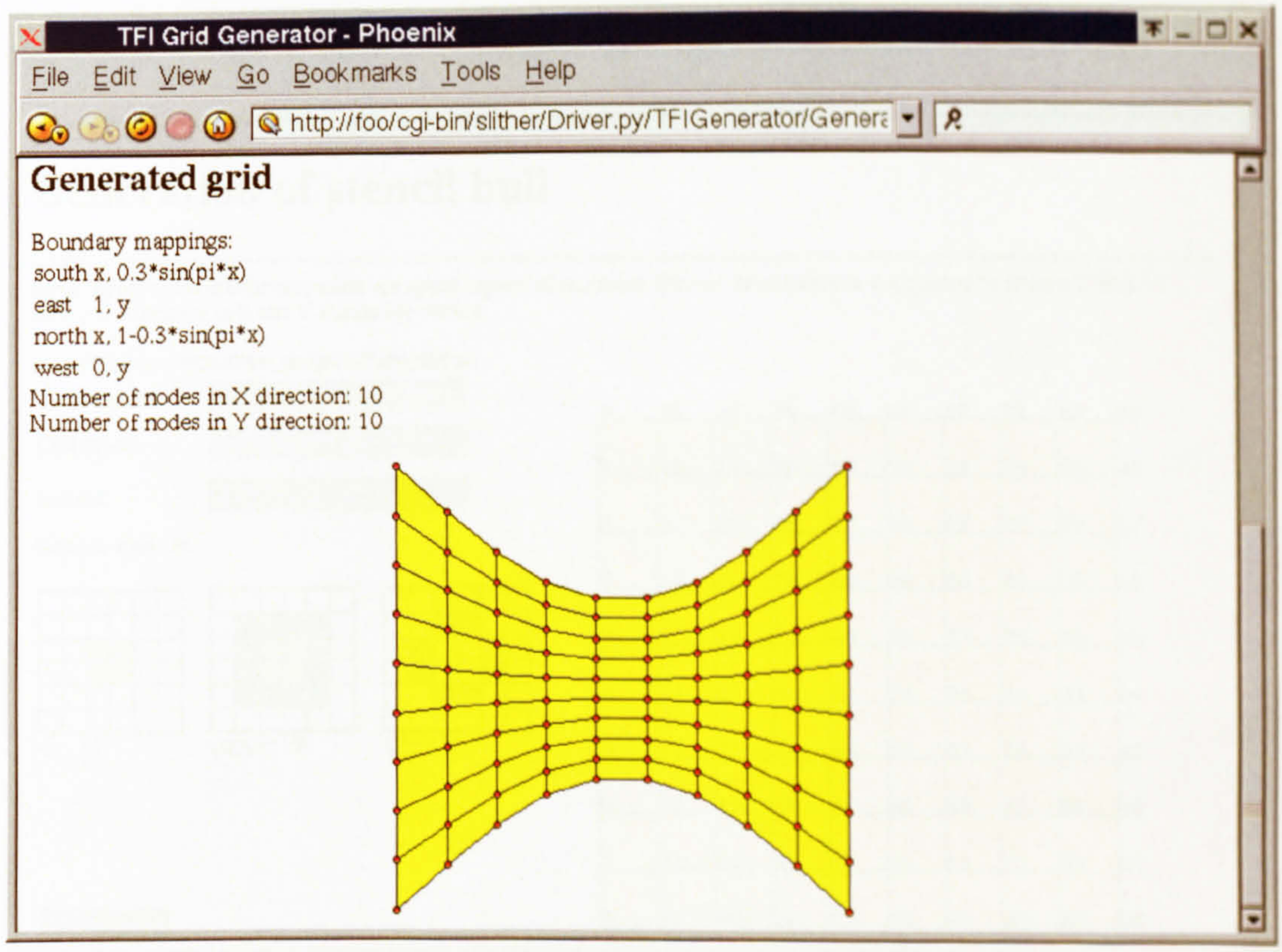


Figure A.4: Results page for the TFI mesh generator service.

A.4 Conclusions

This Chapter did not delve into issues such as security, optimisation or the monitoring of web services. These issues are very important if the service is going to be publicly available. To handle such issues in a proper way, there must be a constant exchange of ideas and feedback between computer science and computational science.

It might not be fully visible from the above presentation, but in the author's opinion web applications and the concept of Grid Computing are the next important step, that will give scientific simulations a new dimension. However, the author also sees the gap between technological advances and educational efforts. The new ideas appear one after another and it might be hard for educators to catch up with that progress. Often, new technologies are explained on general examples or examples far from the readers' experience. Authors do try to give the reader most neutral

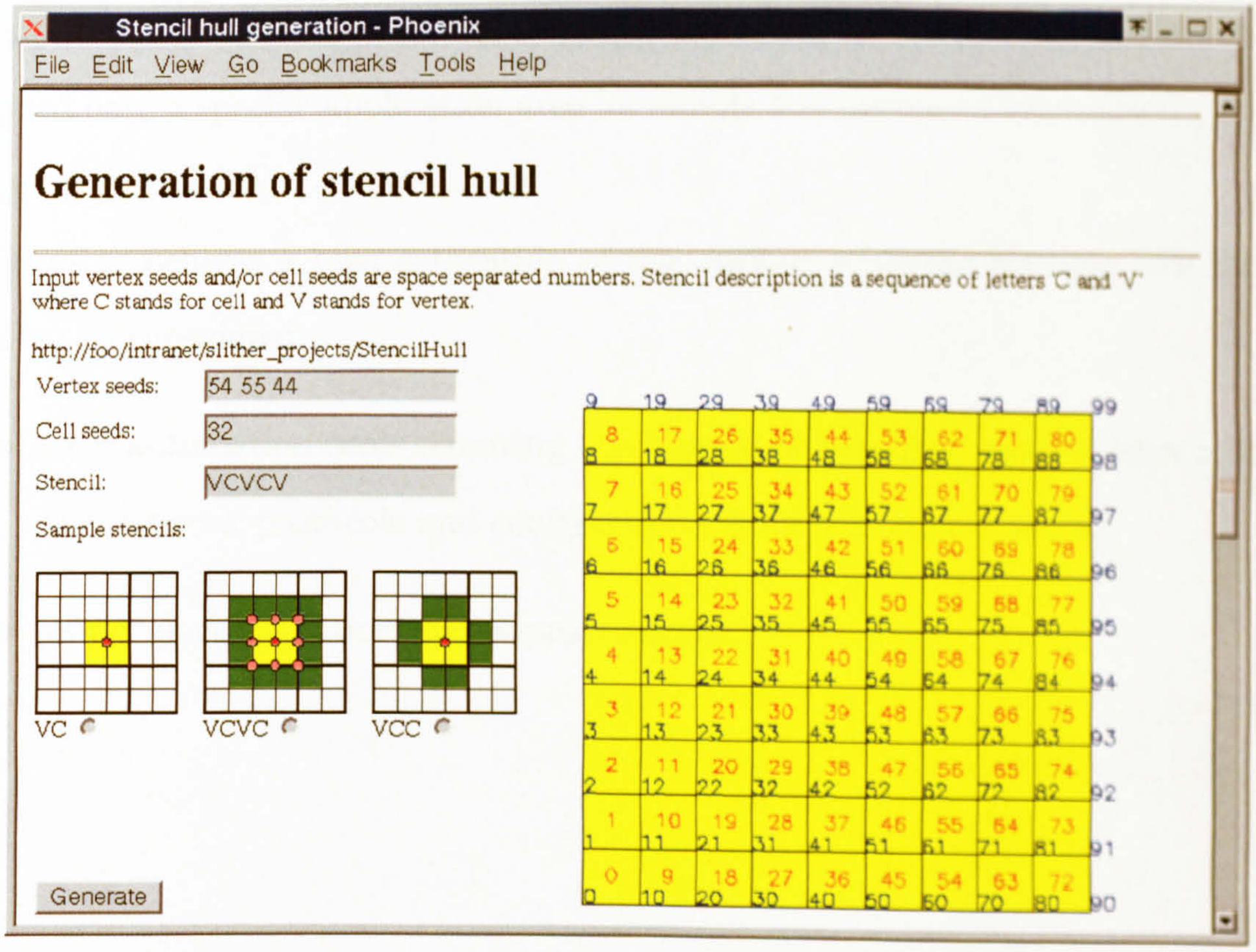


Figure A.5: Data input page for the stencil hull generator service.

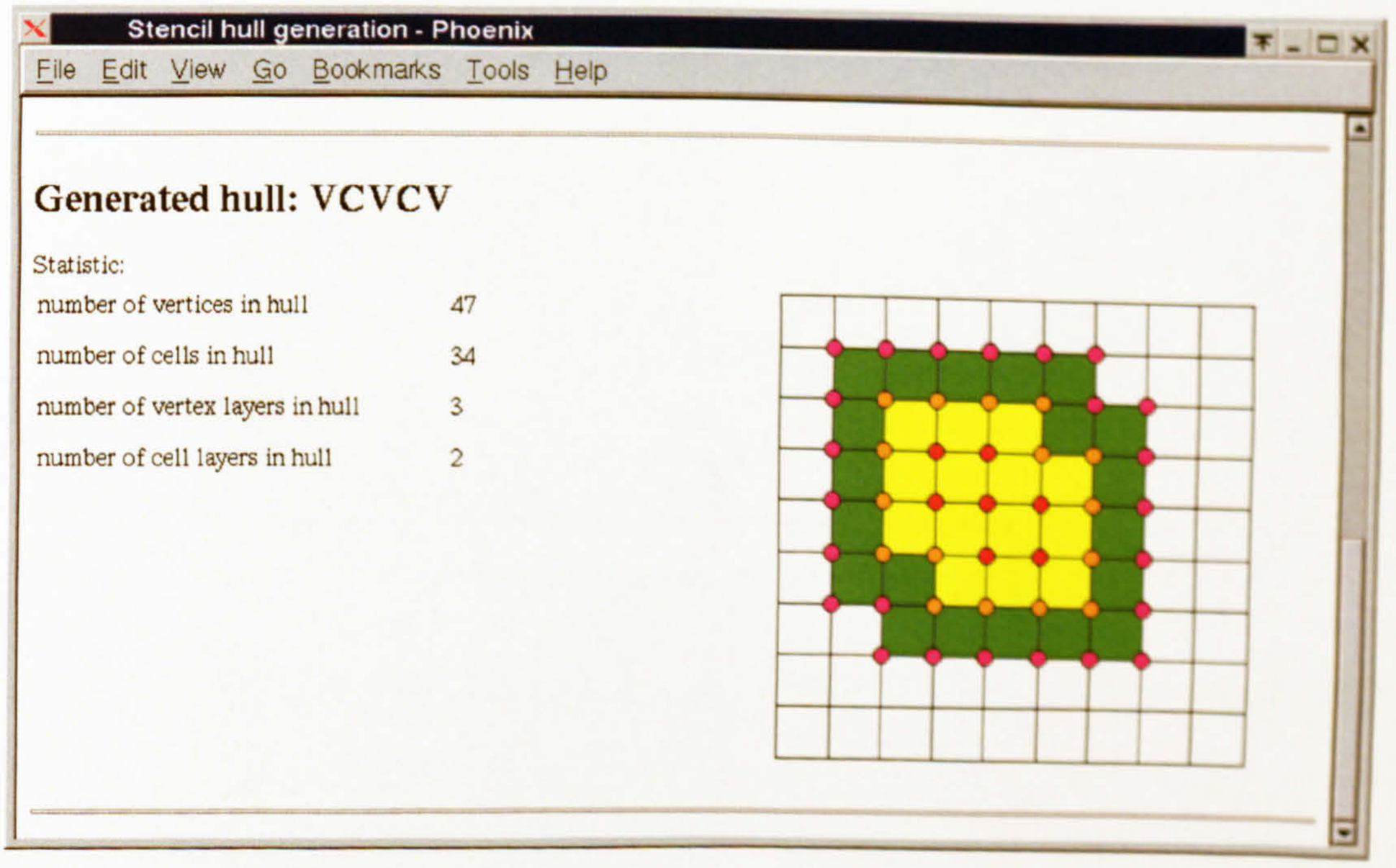


Figure A.6: Results page for the stencil hull generator service.

examples but in some cases it might be difficult to see how new technologies can be mapped onto a specific application area, or simply too boring to read about another shopping portal [79].

Thus to achieve a high saturation of the various scientific services two factors need to be considered:

- good automation tools removing the burden of handling aspects such as security, network protocols and configuration from non-expert users,
- proper documentation and popularisation.

Appendix B

Interface to LPlotter library

```
1  namespace lplotter {
2
3  typedef enum {KEEP, FITX, FITY} aspect_type;
4  typedef enum {LEFT=0, BOTTOM=0, CENTER=1, RIGHT=2, TOP=2} alignment_type;
5  typedef enum {CM, MM, INCH, PT} unit_type;
6  typedef enum {BITMAPPED_GRAPHICS, VECTOR_GRAPHICS, UNKNOWN} graphics_type;
7  typedef enum {S,N,E,W,SE, SW, NE, NW, C} anchor_type;
8
9  double Inch2Unit(double d, unit_type unit);
10 double Unit2Inch(double d, unit_type unit);
11
12 class LPlotter;
13 class Legend;
14
15 class LegendItem {
16   public:
17
18   LegendItem(const char *description, const char *color="black");
19   virtual ~LegendItem();
20   void SetDescription(const char *description);
21   std::string const & GetDescription() const;
22   void SetColor(const char *color);
23   std::string const & GetColor() const;
24 };
25
26 class Legend {
27   public:
28
```

```
29      Legend();
30      virtual ~Legend();
31      typedef std::vector<LegendItem> items_container;
32      typedef items_container::const_iterator items_iterator;
33
34      int AppendItem(LegendItem const& item);
35      int AppendItem(const char* description, const char* color="black");
36      LegendItem GetItem(int at) const ;
37      void SetItem(LegendItem const &item, int at);
38
39      void SetFontSize(double f, unit_type unit=INCH);
40      double GetFontSize(unit_type unit=INCH) const;
41
42      void SetBaselineStretch(double bs);
43      void SetAnchor(anchor_type a);
44      anchor_type GetAnchor() const;
45
46      std::string const & GetTitle() const;
47      void SetTitle(const char* t);
48
49      std::string const & GetColor() const;
50      void SetColor(const char* c);
51
52      void SetFrame(bool f);
53      bool GetFrame() const;
54      void SetFrameOn();
55      void SetFrameOff();
56
57      double GetWidth(LPlotter const &plotter, unit_type unit=INCH) const;
58      double GetHeight(LPlotter const &plotter, unit_type unit=INCH) const;
59
60  };
61  class PlotStyle {
62    public:
63
64      PlotStyle();
65      virtual ~PlotStyle();
66      void SetDefault(std::string paramname, std::string value);
67      std::string colorname;
68      std::string pencolorname;
69      std::string fillcolorname;
70      int filltype;
71      std::string linemod;
```

```
72      int pentype;
73      double linewidth;  // This is viewport line width;
74      double point_radius;
75      unit_type unit;
76      char arrowtype;
77      double arrowwidth;
78      double arrowheight;
79  };
80
81  class LPlotter {
82    public:
83    static LPlotter* New(const char *type="X");
84
85    LPlotter& GetBasePlotter(void);
86
87    void Delete(void);
88
89    void PrintSelf(void);
90
91    void IncrRefCount(void);
92    void DecrRefCount(void);
93
94    int GetRefCount(void);
95
96    bool Show(double x0, double y0, double x1, double y1);
97
98    double viewport2user(double distance, const unit_type unit=INCH) const;
99
100   void viewport2user(const double viewport_x, const double viewport_y,
101                      double *user_x, double *user_y, const unit_type unit=INCH) const;
102
103   double user2viewport(double distance, const unit_type unit=INCH) const;
104
105   void user2viewport(const double user_x, const double user_y,
106                      double *viewport_x, double *viewport_y,
107                      const unit_type unit=INCH) const;
108
109   double GetUserX(const int i) const;
110   double GetUserY(const int i) const;
111
112   void DoubleBufferingOn(void);
113   void SetViewportBorder(double border, const unit_type unit=INCH);
114   double GetViewportBorder(const unit_type unit=INCH) const;
```

```
115
116     void SetViewportLineWidth(double width, const unit_type unit=INCH);
117     double GetViewportLineWidth(const unit_type unit=INCH) const;
118     void SetViewportFontSize(double f, unit_type unit=INCH);
119     double GetViewportFontSize(unit_type unit=INCH) const;
120
121     double GetViewportWidth(const unit_type unit=INCH) const;
122     double GetViewportHeight(const unit_type unit=INCH) const;
123
124     double GetUserWidth() const;
125     double GetUserHeight() const;
126
127     void PlotViewportFrame();
128
129     graphics_type GetGraphicsType() const;
130     bool IsBitmapType() const;
131     bool IsVectorType() const;
132
133     void PrintSelf() const;
134
135     void SetAspectRatio(aspect_type a);
136     aspect_type GetAspectRatio() const;
137
138     void SetAlignment(alignment_type a);
139     alignment_type GetAlignment() const;
140
141     void SetOutputFileName(const char *name);
142     void SetErrorFileName(const char *name);
143
144     const char* GetType() const;
145
146     void SetViewportSize(double width, double height,
147                          const unit_type unit=INCH, const
148                          char *format="a4");
149
150     PlotStyle const & GetStyle();
151     void SetStyle (lplotter::PlotStyle const & s);
152     void UpdateStyle(std::string paramname, std::string value);
153
154     inline void PlotPoint(double x, double y, lplotter::PlotStyle const *styles[1]=NULL);
155
156     void PlotPoint(double *coords,PlotStyle const *styles[1]=NULL);
157
```

```
158     inline void PlotSegment(double x1, double y1, double x2, double y2,
159         PlotStyle const *styles[1]=NULL);
160     void PlotSegment(double *p1, double *p2,PlotStyle const *styles[1]=NULL);
161     inline void PlotTriangle(double x1, double y1, double x2, double y2,
162                              double x3, double y3, PlotStyle const *styles[1]=NULL);
163     void PlotTriangle(double *p1, double *p2, double *p3, PlotStyle const *styles[1]=NULL);
164     inline void PlotQuad(double x1, double y1, double x2, double y2,
165                          double x3, double y3, double x4, double y4, PlotStyle const *styles[
166     void PlotQuad(double *p1, double *p2, double *p3, double *p4,PlotStyle const *styles[1]=N
167
168
169     void vlegend(int x, int y, Legend const &legend);
170     void vflegend(double x, double y, Legend const &legend);
171
172     void flegend(double x, double y, Legend const &legend);
173
174     void legend(int x, int y, Legend const &legend);
175
176     void legend(anchor_type anchor, Legend const &legend,
177                 bool respectBorder=false);
178
179     // Original GNU libplot interface
180     int arc(int xc, int yc, int x0, int y0, int x1, int y1);
181     int box(int x0, int y0, int x1, int y1);
182     int circle(int x, int y, int r);
183     int cont(int x, int y);
184     int erase();
185     int label (const char *s);
186     int line (int x0, int y0, int x1, int y1);
187     int linemod (const char *s);
188     int move (int x, int y);
189     int point (int x, int y);
190     void space (int x0, int y0, int x1, int y1);
191     int alabel (int x_justify, int y_justify, const char *s);
192     int arcrel (int dxc, int dyc, int dx0, int dy0, int dx1, int dy1);
193     int bezier2 (int x0, int y0, int x1, int y1, int x2, int y2);
194     int bezier2rel (int dx0, int dy0, int dx1, int dy1, int dx2, int dy2);
195     int bezier3 (int x0, int y0, int x1, int y1, int x2, int y2,
196                                                  int x3, int y3);
197     int bezier3rel (int dx0, int dy0, int dx1, int dy1, int dx2, int dy2,
198                                                      int dx3, int dy3);
199     int bgcolor (int red, int green, int blue);
200     int bgcolorname (const char *name);
```

```
201     int boxrel (int dx0, int dy0, int dx1, int dy1);
202     int capmod (const char *s);
203     int circlerel (int dx, int dy, int r);
204     int closepath ();
205     int color (int red, int green, int blue);
206     int colorname (const char *name);
207     int contrel (int x, int y);
208     int ellarc (int xc, int yc, int x0, int y0, int x1, int y1);
209     int ellarcrel (int dxc, int dyc, int dx0, int dy0, int dx1, int dy1);
210     int ellipse (int x, int y, int rx, int ry, int angle);
211     int ellipserel (int dx, int dy, int rx, int ry, int angle);
212     int endpath ();
213     int endsubpath ();
214     int fillcolor (int red, int green, int blue);
215     int fillcolorname (const char *name);
216     int fillmod (const char *s);
217     int filltype (int level);
218     int flushpl ();
219     int fontname (const char *s);
220     int fontsize (int size);
221     int havecap (const char *s);
222     int joinmod (const char *s);
223     int labelwidth (const char *s);
224     int linedash (int n, const int *dashes, int offset);
225     int linerel (int dx0, int dy0, int dx1, int dy1);
226     int linewidth (int size);
227     int marker (int x, int y, int type, int size);
228     int markerrel (int dx, int dy, int type, int size);
229     int moverel (int x, int y);
230     int orientation (int direction);
231     int pencolor (int red, int green, int blue);
232     int pencolorname (const char *name);
233     int pentype (int level);
234     int pointrel (int dx, int dy);
235     int restorestate () const;
236     int savestate () const;
237     int textangle (int angle);
238     double ffontname (const char *s);
239     double ffontsize (double size) const;
240     double flabelwidth (const char *s) const;
241     double ftextangle (double angle) const;
242     int farc (double xc, double yc, double x0, double y0, double x1, double y1);
243     int farcrel (double dxc, double dyc, double dx0, double dy0,
```

```
244                     double dx1, double dy1);
245     int fbezier2 (double x0, double y0, double x1, double y1,
246                     double x2, double y2);
247     int fbezier2rel (double dx0, double dy0, double dx1, double dy1,
248                     double dx2, double dy2);
249     int fbezier3 (double x0, double y0, double x1, double y1, double x2,
250                     double y2, double x3, double y3);
251     int fbezier3rel (double dx0, double dy0, double dx1,
252                     double dy1, double dx2, double dy2,
253                     double dx3, double dy3);
254     int fbox (double x0, double y0, double x1, double y1);
255     int fboxrel (double dx0, double dy0, double dx1, double dy1);
256     int fcircle (double x, double y, double r);
257     int fcirclerel (double dx, double dy, double r);
258     int fcont (double x, double y);
259     int fcontrel (double dx, double dy);
260     int fellarc (double xc, double yc, double x0, double y0,
261                     double x1, double y1);
262     int fellarcrel (double dxc, double dyc, double dx0,
263                     double dy0, double dx1, double dy1);
264     int fellipse (double x, double y, double rx, double ry, double angle);
265     int fellipserel (double dx, double dy, double rx,
266                     double ry, double angle);
267     int flinedash (int n, const double *dashes, double offset);
268     int fline (double x0, double y0, double x1, double y1);
269     int flinerel (double dx0, double dy0, double dx1, double dy1);
270     int flinewidth (double size);
271     int fmarker (double x, double y, int type, double size);
272     int fmarkerrel (double dx, double dy, int type, double size);
273     int fmove (double x, double y);
274     int fmoverel (double dx, double dy);
275     int fpoint (double x, double y);
276     int fpointrel (double dx, double dy);
277     int fconcat (double m0, double m1, double m2, double m3,
278                     double m4, double m5);
279     int fmiterlimit (double limit);
280     int frotate (double theta);
281     int fscale (double x, double y);
282     int fsetmatrix (double m0, double m1, double m2, double m3,
283                     double m4, double m5);
284     int ftranslate (double x, double y);
285     int closepl();
286 };
```

```
287  } // namespace lplotter
```

Appendix C

Interface to ONPlot library

```
1  namespace on_plot_2D {
2
3  void PlotArc(lplotter::LPlotter *plotter, ON_Arc const &arc);
4  void PlotCircle(lplotter::LPlotter *plotter, ON_Circle const  &circle);
5  void PlotEllipse(lplotter::LPlotter *plotter, ON_Ellipse const &ellipse);
6  void PlotBezierControlPolygon(lplotter::LPlotter *plotter, ON_BezierCurve const &bezier);
7  void PlotBezierUniform(lplotter::LPlotter *plotter, ON_BezierCurve const &bezier,
8       int npoints=10, bool subpath=false);
9
10 void PlotNurbsControlPolygon(lplotter::LPlotter *plotter, ON_NurbsCurve const &nurbs);
11 void PlotNurbsUniform(lplotter::LPlotter *plotter, ON_NurbsCurve const &nurbs,
12      ON_Interval *domain=NULL, int npoints=10);
13
14 void PlotNurbsUniform(lplotter::LPlotter *plotter, ON_NurbsCurve const &nurbs,
15      double t_begin, double t_end, int npoints=10);
16
17 int PlotNurbsAdaptive(lplotter::LPlotter *plotter, ON_NurbsCurve const &nurbs,
18      double tolerance=0.01, ONC_SegmentSampler::eTestMethod
19                             method=ONC_SegmentSampler::CHORD);
20
21 void PlotMesh(lplotter::LPlotter* plotter, ON_Mesh const &mesh);
22 void PlotPolyline(lplotter::LPlotter* plotter, ON_Polyline const& polyline);
23
24
25 template <class POINT>
26 void PlotTriangle(lplotter::LPlotter *plotter, POINT const& p1,
27                                                POINT const& p2,
28                                                POINT const& p3);
```

```
29  template <class POINT>
30  void PlotQuad(lplotter::LPlotter *plotter, POINT const& p1,
31                                              POINT const& p2,
32                                              POINT const& p3,
33                                              POINT const& p4);
34
35  void lplotter::SetPlotStyle(lplotter::LPlotter &plotter, lplotter::PlotStyle const &style);
36  void lplotter::ResetPlotStyle(lplotter::LPlotter &plotter);
37
38  class ONPlotter  {
39    public:
40      explicit ONPlotter(const char *type);
41      explicit ONPlotter(lplotter::LPlotter *p);
42      ONPlotter();
43
44      lplotter::LPlotter& GetBasePlotter(void);
45
46      int Show(ON_BoundingBox const & bb);
47      int Show(double, double, double, double);
48
49      void PlotArc(ON_Arc const &arc);
50      void PlotCircle(ON_Circle const  &circle, bool endpath_=1);
51      void PlotEllipse(ON_Ellipse const  &ellipse, bool endpath_=1);
52      void PlotBezierControlPolygon(ON_BezierCurve const &bezier);
53      void PlotBezierUniform(ON_BezierCurve const &bezier, int npoints=10, bool subpath=false
54      void PlotNurbsControlPolygon(ON_NurbsCurve const &nurbs);
55      void PlotNurbsUniform(ON_NurbsCurve const &nurbs, ON_Interval *domain=NULL,
56                            int npoints=10);
57      void PlotNurbsUniform(ON_NurbsCurve const &nurbs, double t_begin,
58                            double t_end, int npoints=10);
59      int  PlotNurbsAdaptive(ON_NurbsCurve const &nurbs, double tolerance=0.01);
60      void PlotMesh(ON_Mesh const &mesh);
61      void PlotPolyline(ON_Polyline const& polyline);
62      void PlotBoundingBox(ON_BoundingBox const &bbox, bool show_verts=false,
63          lplotter::PlotStyle const *styles[1]=NULL);
64      void PlotPoint(ON_3dPoint const & point);
65      void PlotPoint(ON_2dPoint const & point);
66      void PlotPoint(ON_3fPoint const & point);
67      void PlotPoint(ON_2fPoint const & point);
68      void PlotPoint(double x, double y);
69      void PlotVector(ON_2dVector const & v, ON_2dPoint const & p);
70      void PlotVector(ON_2fVector const & v, ON_2fPoint const & p);
71      void PlotVector(ON_3dVector const & v, ON_3dPoint const & p);
```

```
72      void PlotVector(ON_3fVector const & v, ON_3fPoint const & p);
73  };
74
75  class  KnotLine {
76    public:
77      enum {HORIZONTAL=0, VERTICAL=1};
78      enum {BOTTOM, RIGHT, TOP, LEFT};
79      KnotLine();
80      void Plot(ONPlotter & onplotter, ON_NurbsCurve const &nurbs);
81      double x,y;
82
83      void SetHorizontal();
84      void SetVertical();
85      void SetOrientation(const int o);
86      int GetOrientation() const;
87      void SetSide(const int s);
88      int GetSide() const;
89      void SetLength(const double l);
90      double GetLength() const;
91      void SetDisplayLineWidth(const double w);
92      double GetDisplayLineWidth() const;
93      void SetTickDisplayLength(const double t);
94      double GetTickDisplayLength() const;
95
96  };
97
98  } // namespace on_plot_2D
```

Appendix D

Interface to cubic mesh generator

```
1  namespace cubic {
2
3  class CubicError : public std::exception {
4    public:
5      CubicError() : msg("Unspecified error");
6      explicit CubicError(std::string message) : msg(message);
7      virtual ~CubicError() throw()
8      virtual const char* what() const throw ();
9    protected:
10     std::string msg;
11 };
12
13 class ParseError : public CubicError {
14   public:
15    ParseError() : CubicError("Unspecified parse error") {}
16    explicit ParseError(std::string message) {
17      msg = "Parse error: " + message;}
18    ~ParseError() throw() {}
19 };
20
21 class InvalidArgumentError : public CubicError {
22   public:
23    InvalidArgumentError() : CubicError("Unspecified parse error") {};
24    explicit InvalidArgumentError(std::string message) {
25      msg = "Argument error: " + message;}
26    ~InvalidArgumentError() throw() {}
27 };
28
```

```
29  class AllocationError : public CubicError {
30    public:
31     AllocationError() : CubicError("Unspecified parse error") {}
32     explicit AllocationError(std::string message) {
33       msg = "Allocation error: " + message;}
34     ~AllocationError() throw() {}
35  };
36
37  class InternalError : public CubicError {
38    public:
39     InternalError() : CubicError("Unspecified parse error") {}
40     explicit InternalError(std::string message) {
41       msg = "Internal error: " + message;}
42     ~InternalError() throw() {}
43
44  };
45
46  class CubicGenerator {
47     private:
48       GEN *gen_;
49       std::string filename_;
50
51     public:
52       typedef enum {DEFAULT_ELEMENT = -1,
53                     LINK1    = E_ELEM_TYPE_LINK1,
54                     TRIANG1  = E_ELEM_TYPE_TRIANG1,
55                     TRIANG3  = E_ELEM_TYPE_TRIANG3,
56                     TRIANG4  = E_ELEM_TYPE_TRIANG4,
57                     TRIANG5  = E_ELEM_TYPE_TRIANG5,
58                     QUAD1    = E_ELEM_TYPE_QUAD1} elem_type;
59
60
61       explicit CubicGenerator(std::string filename);
62       CubicGenerator(int nnodes, int nsurfaces, int nlinks);
63       MESH* GenerateMesh() {return e_CubicGenerateMesh(gen_);}
64       ~CubicGenerator();
65
66       void SetNumOfMaterials(const int i);
67       void SetNumOfBCNodes(const int i);
68       void SetNumOfBCSides(const int i);
69
70       void SetXDivision(int patchIndex, int div) {
71       void SetYDivision(int patchIndex, int div) {
```

```
72        void SetTDivision(int patchIndex, int div) {
73        void SetDivision(int patchIndex, int dir, int div) throw (InternalError);
74
75        void SetLinearPatch(int patchIndex, int shape, int *nodes,
76                                                elem_type et=DEFAULT_ELEMENT);
77        void SetParametricPatch(int patchIndex, int shape, int *nodes,
78                            double *derivatives, elem_type et=DEFAULT_ELEMENT);
79        void SetInterpolatingPatch(int patchIndex, int shape,
80                                   int *nodes, elem_type et=DEFAULT_ELEMENT);
81
82        void SetPatchElementType(int patchIndex, elem_type et);
83
84        void SetNodeCoords(int nodeIndex, double const *coords);
85        void SetNodeCoords(int nodeIndex, const double x, const double y,
86                                          const double z=0.0);
87        void Save(std::string filename) const;
88  };
89
90  } // namespace cubic
```

Appendix E

Implementation of GtsGLRenderer class

```
1  from OpenGL.GL import *
2  from OpenGL.GLU import *
3  import math
4  from GTS.Objects.surfaces import *
5  from GTS.Objects.vertices import *
6  from GTS.Objects.triangles import *
7  from GTS.GL.gl import *
8
9  class GtsGLRenderer(object):
10     """Map GtsSufrace to OpenGL display list"""
11     def __init__(self, surface):
12         super(GtsGLRenderer,self).__init__()
13         self.__surface = surface
14         self.__gldl = []
15
16     def __delete__(self):
17         self.__clear_dl()
18
19     def __clear_dl(self):
20         """Removes all display lists
21         """
22         if self.__gldl is not []:
23             err = glDeleteLists(self.__gldl[0], len(self.__gldl))
24             self.__gldl = []
25
```

```
26      def __build_gldl(self):
27          if self.__gldl != []:
28              raise RuntimeError, \
29                  "GtsRenderer: Internal error, display list not empty"
30          nlists = 3
31          dl = glGenLists(nlists); # display list for vertices, edges, faces
32          self.__gldl = range(dl, dl+nlists)
33          self.__build_faces_dl(self.__gldl[2])
34          self.__build_edges_dl(self.__gldl[1])
35
36      def __build_single_face(self, face, data):
37          [v1, v2, v3] = gts_triangle_vertices(face);
38          glVertex3f(v1.p.x,v1.p.y,v1.p.z)
39          glVertex3f(v2.p.x,v2.p.y,v2.p.z)
40          glVertex3f(v3.p.x,v3.p.y,v3.p.z)
41          return 0
42
43      def __build_faces_dl(self, dl):
44          glNewList(dl, GL_COMPILE);
45          glEnable(GL_AUTO_NORMAL)
46          glMaterialfv(GL_FRONT, GL_AMBIENT, [1, 0, 0, 0])
47          glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, [0, 0, 0, 0])
48          glMaterialfv(GL_BACK, GL_AMBIENT, [0, 1, 0, 0])
49          glLightModeli(GL_LIGHT_MODEL_TWO_SIDE ,1)
50          glBegin(GL_TRIANGLES)
51          gts_surface_foreach_face(self.__surface, self.__build_single_face,
52          None)
53          glEnd()
54          glEndList()
55
56      def __build_edges_dl_old(self, dl):
57          glNewList(dl, GL_COMPILE)
58          glDisable(GL_LIGHTING)
59          glLineWidth(4)
60          glColor3f(0,0,0)
61          glBegin(GL_LINES)
62          glVertex3f(0,0,0)
63          glVertex3f(1,2,1)
64          glVertex3f(1,2,1)
65          glVertex3f(0,0,1)
66          glVertex3f(0,0,1)
67          glVertex3f(0,0,0)
68          glEnd()
```

```
69          glEnable(GL_LIGHTING)
70          glEndList()
71
72      def __build_edges_dl(self, dl):
73          glNewList(dl, GL_COMPILE)
74          glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, [0, 0, 0, 0])
75          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, [0, 0, 0, 0])
76          eprop = gts_gl_edge_prop_new()
77          eprop.type=GTS_GL_EDGE_AS_CYLINDER
78          glLineWidth(2)
79          eprop.radius=0.05
80          eprop.r_resolution=8
81          self.ec=0
82          gts_surface_foreach_edge(self.__surface, self.__build_single_edge,
83          eprop)
84          print eprop.radius
85          glEndList()
86
87      def __build_single_edge(self, edge, prop):
88          self.ec+=1
89          gts_gl_edge(edge, prop)
90          return 0
91
92      def __build_edges_dl_cyl(self, dl):
93          self.__quadric =  gluNewQuadric()
94          gluQuadricNormals(self.__quadric, GLU_SMOOTH)
95          glVertex3f(0,0,0)
96          glVertex3f(1,2,1)
97          l = math.sqrt(5);
98          cc = [1.0/l,2.0/l,1.0/l]
99          cc1 = math.sqrt(cc[0]**2 + cc[1]**2)
100         r = 0.1
101         nr = 3
102         nz = 1
103         h = 2
104         m1 = [cc[0], cc[1], 0 ,0,
105               -cc[1], cc[0], 0, 0,
106               0,      0,     1, 0,
107               0,      0,     0, 1]
108         m2 = [cc1, 0, cc[2] ,0,
109               0,   1, 0, 0,
110              -cc[2],0,cc1, 0,
111              0 ,0 ,0, 1]
```

```
112             glNewList(dl, GL_COMPILE)
113             glPushMatrix()
114             glMultMatrixd(m1)
115             glMultMatrixd(m2)
116             glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, [0, 0, 0, 0])
117             glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, [0, 0, 0, 0])
118             gluCylinder(self.__quadric,r,r, l, nr, nz)
119             glEnable(GL_LIGHTING)
120             glPopMatrix()
121             glEndList()
122
123         def Render(self):
124             if self.__gldl == []:
125                 self.__build_gldl()
126             glCallList(self.__gldl[2])
127             glCallList(self.__gldl[1])
128
129         def Update(self):
130             pass
```